VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

DIPLOMOVÁ PRÁCE

Brno, 2019 Bc. Kateřina Bijotová

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ
DEPARTMENT OF TELECOMMUNICATIONS

PARALELIZACE NÁROČNÝCH ÚLOH REKONSTRUKCE V DYNAMICKÉ MAGNETICKÉ REZONANCI
PARALLELIZATION OF COMPLEX TASKS IN RECONSTRUCTION OF DYNAMIC MAGNETIC RESONANCE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Kateřina Bijotová**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Jan Mašek, Ph.D.**
SUPERVISOR

BRNO 2019

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní obor **Telekomunikační a informační technika**
Ústav telekomunikací

Studentka: Bc. Kateřina Bijotová ***ID:*** 147672
Ročník: 2 ***Akademický rok:*** 2018/19

NÁZEV TÉMATU:

Paralelizace náročných úloh rekonstrukce v dynamické magnetické rezonanci

POKYNY PRO VYPRACOVÁNÍ:

Cílem diplomové práce bude prostudování technologie magnetické rezonance a následná optimalizace rekonstrukčního modelu s pomocí grafického akcelerátoru. Stěžejní část práce se bude zabývat optimalizací výpočetně náročných částí rekonstrukčního modelu v Matlabu s pomocí funkce "gpuArray" a dále bude na základě kódu v Matlabu vytvořena a otestována funkce NUFFT s pomocí technologie CUDA a knihovy Jcuda. Takto vytvořená optimalizovaná funkce bude spouštěna z původního kódu v Matlabu a jednotlivé výsledky zrychlení výpočtů budou srovnány a vykresleny do grafů. Dále bude provedeno vyhodnocení na reálných datech z MR skeneru.

DOPORUČENÁ LITERATURA:

[1] LUSTIG, M., D.L. DONOHO, J.M. SANTOS a J.M. PAULY. Compressed Sensing MRI. Signal Processing Magazine, IEEE. USA: IEEE, 0803, 25(2), 72-82. DOI: 10.1109/MSP.2007.914728. ISSN 1053-5888.

[2] HAGER, Georg a Gerhard WELLEIN. Introduction to high performance computing for scientists and engineers. Boca Raton: CRC Press, c2011. Chapman & Hall/CRC computational science series, 7. ISBN 978-1-4398-1-92-4.

Termín zadání: 1.2.2019 ***Termín odevzdání:*** 16.5.2019

Vedoucí práce: Ing. Jan Mašek, Ph.D.
Konzultant:

prof. Ing. Jiří Mišurec, CSc.
předseda oborové rady

ABSTRAKT

Tato diplomová práce se zabývá paralelizací náročných úloh rekonstrukce v dynamické magnetické rezonanci. Popisuje základní princip magnetické rezonance a její souvislost s Fourierovou transformací. Zabývá se rozdílem mezi statickou a dynamickou rekonstrukcí obrazu z magnetické rezonance. Rozebírá algoritmus SVD a jeho použití při rekonstrukci zobrazování magnetickou rezonancí. Uvádí princip a význam paralelních výpočtů při zobrazování magnetickou rezonancí a dále popisuje technologii CUDA. Dále práce obsahuje popis a vypracování implementace rekonstrukčního modelu v jazyce MATLAB a jazyce Java, jež byly optimalizovány knihovnou JCuda v případě Java implementace a funkcí gpuArray pro MATLAB implementaci.

KLÍČOVÁ SLOVA

Zobrazování magnetickou rezonancí, NUFFT, proximální algoritmy, paralelní výpočty, gpuArray, CUDA.

ABSTRAKT

This thesis deals with parallelization of complex tasks in reconstruction of dynamic magnetic resonance. It describes the basic principle of magnetic resonance and its relation to Fourier transform. It deals with the difference between static and dynamic magnetic resonance image reconstruction. It analyzes SVD algorithm and its use in magnetic resonance image reconstruction. It presents the principles and the importance of parallel computing in magnetic resonance imaging and describes CUDA technology. The thesis also contains a description and execution of the implementation of the reconstruction model in MATLAB and Java programming language which were optimized by JCuda library for Java implementation and gpuArray function in case of MATLAB implementation.

KLÍČOVÁ SLOVA

Magnetic resonance imaging, NUFFT, proximal algorithms, parallel computing, gpuArray, CUDA.

BIJOTOVÁ, Kateřina. *Paralelizace náročných úloh rekonstrukce v dynamické magnetické rezonanci*. Brno, Rok, 63 s. Diplomová práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací. Vedoucí práce: Ing. Jan Mašek, Ph.D.

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma „Paralelizace náročných úloh rekonstrukce v dynamické magnetické rezonanci“ jsem vypracovala samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autorka uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušila autorská práva třetích osob, zejména jsem nezasáhla nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědoma následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autorky

PODĚKOVÁNÍ

Ráda bych poděkovala vedoucímu diplomové práce panu Ing. Janu Maškovi, Ph.D. za konzultace a návrhy k práci a také panu doc. Ing. Pavlovi Rajmicovi, Ph.D. za odborné rady a vysvětlení.

Brno ....

podpis autorky

Faculty of Electrical Engineering
and Communication
Brno University of Technology
Purkynova 118, CZ-61200 Brno
Czech Republic

http://www.six.feec.vutbr.cz

PODĚKOVÁNÍ

Výzkum popsaný v této diplomové práci byl realizován v laboratořích podpořených z projektu SIX; registrační číslo CZ.1.05/2.1.00/03.0072, operační program Výzkum a vývoj pro inovace.

Brno ....

podpis autorky

EVROPSKÁ UNIE
EVROPSKÝ FOND PRO REGIONÁLNÍ ROZVOJ
INVESTICE DO VAŠÍ BUDOUCNOSTI

OP Výzkum a vývoj
pro inovace

Obsah

Úvod **12**

1 Magnetická rezonance **13**
1.1 Základní princip . . . 13
1.1.1 Nukleární magnetický moment . . . 13
1.1.2 Larmorova rovnice . . . 13
1.1.3 Návrat jader do původního stavu . . . 14
1.1.4 Základní rovnice MRI . . . 14
1.1.5 Problémy MRI . . . 15
1.2 Souvislost s Fourierovou transformací . . . 15
1.2.1 Prostorové kódování . . . 15
1.2.2 Použití Fourierovy transformace . . . 15
1.3 Trajektorie v k-prostoru . . . 16
1.3.1 K-prostor . . . 16
1.3.2 Trajektorie . . . 16

2 Rekonstrukce obrazu z MR **20**
2.1 Statická rekonstrukce . . . 20
2.1.1 Základní metody rekonstrukce . . . 20
2.1.2 Regularizace metod rekonstrukce . . . 20
2.2 Dynamická rekonstrukce . . . 21
2.2.1 Proximální algoritmy . . . 21

3 Algoritmus SVD **23**
3.1 Úvod . . . 23
3.2 SVD . . . 23
3.3 Použití algoritmu . . . 23

4 Paralelizace výpočtů **25**
4.1 Význam paralelních výpočtů v MRI . . . 25
4.2 Princip . . . 25
4.2.1 Typy paralelizace . . . 25
4.2.2 Rozdíl mezi GPU a CPU . . . 26
4.2.3 Hierarchie pamětí GPU NVIDIA . . . 27
4.3 CUDA . . . 29
4.3.1 Úvod . . . 29
4.3.2 Popis . . . 29
4.3.3 Pojmy v prostředí CUDA . . . 30

5 Praktická část **31**
5.1 Hardware 31
5.1.1 GPU 31
5.1.2 CPU 31
5.2 Software 32
5.2.1 Vývojové prostředí 32
5.2.2 JCuda 32
5.2.3 gpuArray 33
5.3 Vstupní data MRI 33
5.4 Popis rekonstrukčního modelu v MATLABu 34
5.4.1 Příprava dat 34
5.4.2 Inicializace 35
5.4.3 FFT 35
5.4.4 SVD 35
5.4.5 Iterace 36
5.5 Blokové schéma řešení s paralelními výpočty pomocí nástroje JCuda . 36
5.6 Testování algoritmu SVD 37
5.6.1 Použité implementace algoritmu 38
5.6.2 Naměřené hodnoty 39
5.6.3 Vyhodnocení výsledků 39
5.7 Implementace modelu v MATLABu s pomocí funkce gpuArray 40
5.7.1 Popis implementace 40
5.7.2 Naměřené hodnoty 41
5.7.3 Vyhodnocení výsledků 42
5.8 Implementace náročných úloh v jazyce Java 43
5.9 Optimalizace funkce NUFFT pomocí knihovny JCuda 45
5.9.1 Implementace NUFFT 45
5.9.2 Naměřené délky výpočtů NUFFT 46
5.9.3 Vyhodnocení výsledků 46

6 Závěr **49**

Literatura **51**

Seznam symbolů, veličin a zkratek **54**

Seznam příloh **55**

A Proximálně gradientní algoritmus **56**

B Výsledky měření rychlosti výpočtů implementace rekonstrukčního modelu s pomocí gpuArray **57**

B.1 Naměřené hodnoty délky výpočtů s pomocí gpuArray 57

B.2 Grafická zobrazení srovnání rychlosti výpočtů s pomocí gpuArray a zadaného rekonstrukčního modelu 58

C Ukázkové kódy **60**

D Obsah přiloženého CD **63**

Seznam obrázků

1.1 Zobrazený precesní pohyb vzniklý působením vnějšího magnetického pole označeného B_0. 14
1.2 Kartézská trajektorie. 17
1.3 Trajektorie cikcak. 18
1.4 Radiální trajektorie. 19
1.5 Spirálová trajektorie. 19
4.1 Srovnání architektury CPU (vlevo) a GPU (vpravo). 26
4.2 Hierarchický model paměti GPU NVIDIA. 27
5.1 Fáze programu rekonstrukce MRI v MATLABu. 34
5.2 Volání funkcí ve fázi inicializace rekonstrukčního modelu. 36
5.3 Volání funkcí ve fázi FFT rekonstrukčního modelu - první část gradientního kroku. 37
5.4 Blokové schéma zpracování dat. 38
5.5 Grafické srovnání rychlosti výpočtu algoritmu SVD 40
5.6 Grafické srovnání délky výpočtu inicializace za použití GPU a CPU. . 42
5.7 Grafické srovnání délky výpočtu FFT za použití GPU a CPU. 43
5.8 Volání metod v implementaci první části gradientního kroku v jazyce Java. 44
5.9 Grafické srovnání délky výpočtu funkce NUFFT v různých implementacích pro data s 12 projekcemi na snímek. 46
5.10 Grafické srovnání délky výpočtu funkce NUFFT v různých implementacích pro data s 12 projekcemi na snímek. 47
B.1 Grafické srovnání rychlosti výpočtu inicializace za použití GPU a CPU. 58
B.2 Grafické srovnání rychlosti výpočtu inicializace za použití GPU a CPU. 59
B.3 Grafické srovnání rychlosti výpočtu iterace za použití GPU a CPU. . 59

Seznam tabulek

5.1 Výběr z přehledu technických informací GPU primární grafické karty NVIDIA TITAN Xp [16]. 31
5.2 Výběr z přehledu technických informací GPU sekundární grafické karty NVIDIA GTX 750 Ti [16]. 31
5.3 Výběr z přehledu technických informací CPU a RAM. 32
5.4 Srovnání rychlosti provedení algoritmu SVD. 39
5.5 Srovnání průměrné doby výpočtů jednotlivých fází rekonstrukčního modelu v MATLABu mezi původním programem a optimalizovanou implementací na GPU. 41
B.1 Srovnání výpočtů fáze inicializace rekonstrukčního modelu v MATLABu mezi původním programem na CPU a optimalizovanou implementací na GPU. 57
B.2 Srovnání výpočtů fáze FFT rekonstrukčního modelu v MATLABu mezi původním programem na CPU a optimalizovanou implementací na GPU. 57
B.3 Srovnání výpočtů fáze SVD rekonstrukčního modelu v MATLABu mezi původním programem na CPU a optimalizovanou implementací na GPU. 58
B.4 Srovnání výpočtů fáze iterace rekonstrukčního modelu v MATLABu mezi původním programem na CPU a optimalizovanou implementací na GPU. 58

Úvod

Porozumění mozkovým funkcím a zaměření souvisejících funkčních oblastí mozku je důležitým krokem v lidském poznání a předmětem neurověd po celém světě. Jednou z předních metod, které se na mapování mozku zaměřují, je zobrazování magnetickou rezonancí – Magnetic Resonance Imaging, tedy zkráceně MRI, a to jak v oboru vědeckého výzkumu, tak v lékařské praxi. Nejčastější využití je v rámci předoperačního vyšetření pacientů.

Největším úskalím při využití MRI jsou vyšší časové nároky oproti jiným diagnostickým metodám, dané složitými výpočty při zpracování dostatečného objemu dat k vytvoření obrazu. Za účelem tyto nároky snížit vznikla potřeba zefektivnit úlohy rekonstrukce v magnetické rezonanci. Řešením této otázky je využít paralelního zpracování výpočtů oproti klasickému sekvenčnímu způsobu výpočtu, a tedy celý proces urychlit.

Hlavním přínosem práce je vypracování implementace časově náročného algoritmu SVD (rozkladu na singulární hodnoty) a funkce NUFFT (neuniformní rychlá Fourierova transformace), které využívají paralelních výpočtů na grafické kartě. Paralelizace výpočtů má za cíl snížit časové nároky úlohy.

Práce je členěna následovně: v první kapitole je popsán princip magnetické rezonance, využití Fourierovy transformace v MRI a představeny trajektorie pro získávání dat. Kapitola druhá vysvětluje podstatu rekonstrukce obrazu z magnetické rezonance, a to rekonstrukci statickou a dynamickou, a představuje proximální algoritmy. Kapitola třetí rozebírá náročnou úlohu rozkladu na singulární hodnoty pro rekonstrukci v magnetické rezonanci. Předmětem čtvrté kapitoly jsou paralelní výpočty, princip zpracování dat na grafickém procesoru grafické karty a vývojová platforma CUDA – Compute Unified Device Architecture.

Výstupem diplomové práce je praktická část, jež je popsána v následném oddílu práce. V páté kapitole jsou představeny hardwarové a softwarové prostředky, použité k realizaci paralelizovaného výpočtu na grafickém procesoru grafické karty, vstupní data MRI, popis rekonstručního modelu v jazyce MATLAB. Dále se tato část práce zaměřuje na testování algoritmu SVD, implementace modelu v jazyce MATLAB s pomocí funkce gpuArray, implementace funkce NUFFT v jazyce Java a její optimalizace pomocí knihovny JCuda a technologie CUDA. Následně jsou prezentovány, srovnány a diskutovány výsledky testů paralelních a neparalelních výpočtů zmíněných implementací úloh.

1 Magnetická rezonance

1.1 Základní princip

Na rozdíl od dalších zobrazovacích metod v medicíně je magnetická rezonance založena na detekci elektromagnetických vln vyzařovaných tělem. Tento zachycený signál je pak interpretován a zpracován pro vytvoření obrazu lidského těla. Tato část práce popisuje základní fyzikální principy a charakteristiky v zobrazování magnetickou rezonancí a byly v ní použity zdroje [1], [2] a [3].

1.1.1 Nukleární magnetický moment

Každé jádro atomu díky své rotaci kolem imaginární osy (neboli spinu jádra) vytváří magnetické pole, a tuto schopnost nazýváme magnetický moment. Nukleony, které nejsou elektricky neutrální, tj. protony, generují pole silnější, tedy mají větší magnetický moment.

Kromě schopnosti vytváření vlastního magnetického pole ale také magnetický moment vyjadřuje sílu interakce s vnějším magnetickým polem. Orientace vektoru magnetického momentu je v případě, kdy na jádro nepůsobí vnější magnetické pole, náhodná. Působením homogenního vnějšího magnetického pole však dochází k orientaci vektoru magnetického momentu, neboli magnetického dipólu, atomového jádra ve směru působení pole.

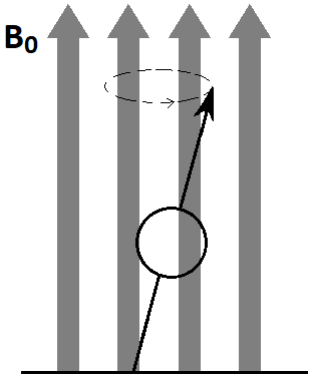
Hlavním chemickým prvkem, který pro zobrazování magnetickou rezonancí – Magnetic Resonance Imaging využíváme, je vodík ^{1}H obsažený ve vodě, která tvoří více než polovinu lidského těla, v tuku a dalších organických molekulách. Působením homogenního magnetického pole B_0 na určitou hmotu však reálně nedocílíme orientace magnetického dipólu všech přítomných jader, ale pouze části. Výsledkem této pře-orientace dipólů je polarizace hmoty. Kromě polarizace také dochází k precesnímu pohybu částic, viz obr. 1.1, neboli rotaci okolo osy souběžné s indukčními čarami působícího pole B_0.

1.1.2 Larmorova rovnice

Úhlový kmitočet precesního pohybu je dán Larmorovou rovnicí (1.1) a nazýváme jej Larmorovou frekvencí [1]:

$$f_0 = \gamma \cdot B_0, \tag{1.1}$$

kde f_0 je Larmorova frekvence vyjádřena pomocí gyromagentického poměru γ (konstanty dané pro jednotlivé nuklidy uváděné v MHz/T) a magnetická indukce pole B_0 udávané v řádu jednotek tesla.



Obr. 1.1: Zobrazený precesní pohyb vzniklý působením vnějšího magnetického pole označeného B_0.

1.1.3 Návrat jader do původního stavu

Pro generování homogenního magnetického pole používáme při MRI vysílání vysokofrekvenčních, neboli RF z anglického Radio Frequency, pulsů o Larmorově frekvenci k pacientovi pomocí vysílacích RF cívek. Vysílané pulsy musí být na rezonančním kmitočtu jader vodíku, aby docházelo k precesnímu pohybu všech jader vodíku o Larmorově frekvenci. Dochází tedy k fázové koherenci, přičemž u některých jader dochází navíc k přechodu do stavu s vyšší energií, tedy k jejich tzv. excitaci.

Po přerušení vysílání RF pulsů jádra ztrácí fázovou koherenci (jádra se odlišují kmitočtem precesního pohybu) a navrací se do původního rovnovážného stavu. Tomuto návratu říkáme relaxace T_2 neboli typu spin–spin. Následkem relaxace se indukuje v přijímací cívce elektrický proud, který představuje signál vyzařovaný magnetizovanými jádry vodíku. Tento signál nazýváme FID – Free induction decay a s postupující relaxací vlivem ztráty fázové koherence signál exponenciálně slábne. Časová konstanta, která tento proces popisuje, se nazývá relaxační čas T_2.

Časová konstanta popisující rychlost návratu magnetických dipólů zpět do původní rovnovážné orientace vlivem interakce spinu se svým okolím je relaxační čas T_1. Tento návrat nazýváme relaxace spin–mřížka či T_1. Obě relaxace, T_1 i T_2 jsou považovány za vzájemně nezávislé souběžně probíhající jevy.

1.1.4 Základní rovnice MRI

Blochova rovnice 1.2 spojuje vyjádření magnetizace působením magnetického pole a následné relaxace do jednoho výrazu [4]:

$$\frac{\mathrm{d}\mathbf{M}}{\mathrm{d}t} = \gamma\mathbf{M} \times \mathbf{B} - \frac{M_x\mathbf{i} + M_y\mathbf{j}}{\mathrm{T}_2} - \frac{M_z - M_0}{\mathrm{T}_1}\mathbf{k}, \tag{1.2}$$

kde $\mathbf{M} = (M_x, M_y, M_z)^T$ vyjadřuje lokální magnetizaci a M_0 rovnovážnou magnetizaci, B odpovídá působícímu magnetickému poli. Výraz $\boldsymbol{\gamma}\mathbf{M} \times \mathbf{B}$ vyjadřuje vektorový součin vektorů magnetizace a magnetické indukce, vektory **i**, **j** a **k** pak určují orientaci v prostoru snímku.

1.1.5 Problémy MRI

Hlavní nevýhodou využití MRI je často dlouhá doba skenování pacienta potřebná ke sběru dostatečného objemu dat pro vytvoření výsledného obrazu. V tomto směru se ukazuje nedostatečnost MRI oproti jiným diagnostickým metodám (počítačová tomografie (CT), pozitronová emisní tomografie (PET) či ultrazvuk), které tak dlouhou dobu pro zachytávání informací nepotřebují. Prodlužování vyšetření zvyšuje nepohodlí pacienta ale také riziko, že se skenovaná osoba pohne a tím zapříčiní nepřesnosti získaného obrazu. Je tedy snaha snížit dobu potřebnou ke sběru dat.

1.2 Souvislost s Fourierovou transformací

Tato podkapitola čerpá ze zdrojů [5] a [6] a nastiňuje využití Fourierovy transformace v zobrazování magnetickou rezonancí.

1.2.1 Prostorové kódování

V MRI průchod elektrického proudu přijímajícími gradientními cívkami způsobí odchylky v magnetickém poli, které odpovídají pozici p snímaného bodu. Působením těchto odchylek (neboli gradientních magnetických polí) mají signály snímané tkáně kmitočet, který je funkcí pozice p [5]:

$$f(p) = \boldsymbol{\gamma} \cdot B(p) = \boldsymbol{\gamma} \cdot (B_0 + G \cdot p), \tag{1.3}$$

kde B je magnetické pole vyjádřeno jako funkce pozice p, B_0 je magnetická indukce statického magnetického pole a G je gradientní magnetické pole.

Pokud by platilo $G = 0$, tedy v případě působení homogenního magnetického pole, signály na všech pozicích by měly stejný kmitočet i průměrnou fázi. V tomto případě bychom nebyli schopni určit, na které pozici v tkáni je signál nejsilnější. Každý bod v tkáni je totiž reprezentován unikátní kombinací odchylek (gradientů) v magnetickém poli, tedy MRI signály jsou jednoznačně prostorově kódovány.

1.2.2 Použití Fourierovy transformace

Gradientní cívky zachycují variace v magnetickém poli v čase, které reprezentují kombinaci signálů o mnoha různých kmitočtech (vycházejících z různých pozic). Ta-

kovýto složený signál je poté aplikováním Fourierovy transformace (FT) rozdělen na jeho jednotlivé kmitočty. FT takto zajišťuje překlad změn napětí na cívce jakožto funkci časově závislých prostorově-frekvenčních souřadnic k_x a k_y v oblasti k-prostoru na intenzitu obrazu odpovídající funkci prostorových souřadnic x a y v obrazovém prostoru. Obě zobrazení představují tutéž informaci v jednotlivých prostorech.

1.3 Trajektorie v k-prostoru

U zobrazování MRI neprobíhá sběr informací o snímaném obrazu napřímo. MRI data jsou získávána v takzvaném k-prostoru, který odpovídá spektrální (neboli také prostorově-kmitočtové) oblasti. V této části práce byly použity zdroje [5], [6], [7], [8] a [9].

1.3.1 K-prostor

Demodulovaný MR signál v časovém okamžiku t je ekvivalentní jedné hodnotě trojrozměrné Fourierovy transformace okamžité magnetizace distribuované ve hmotě, na kterou působíme, v prostorovém kmitočtu daném $\mathbf{k}(t)$, tudíž platí [7]:

$$\mathbf{S}(t) \propto \mathrm{FT}_{3\mathrm{D}}\{M_{xy}(\mathbf{r}, t) \| (\mathbf{k}(t))\}. \tag{1.4}$$

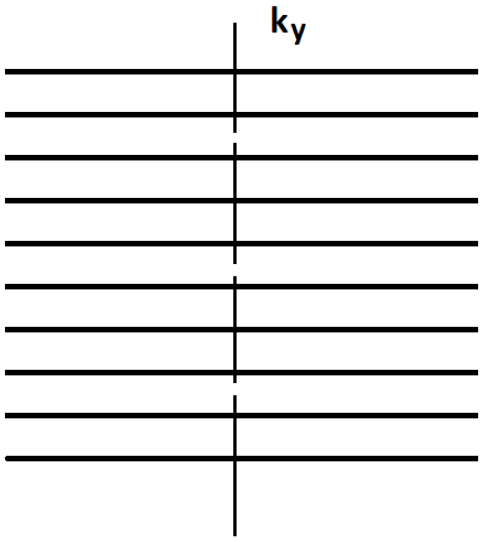
Tento obecný signálový model může být aplikován na jakoukoli MRI metodu měření. Uvedený vzorec 1.4 vyjadřuje, že hodnoty signálu jsou postupně přiřazeny k-prostoru, kde odpovídají jednotlivým hodnotám spektrálního zobrazení skenované oblasti. Symbol $\propto$ značí přímou úměru mezi oběma stranami rovnice. Trajektorie v k-prostoru je určována gradientními poli působícími v době odečítání informací.

1.3.2 Trajektorie

Existuje mnoho způsobů, jakými můžeme sbírat data potřebná pro rekonstrukci obrazu. V jakém pořadí získáváme body k_x a k_y popisují trajektorie v k-prostoru. Nejužívanější trajektorie jsou kartézské, typu cikcak, radiální a spirálové.

Kartézská trajektorie

Tento typ trajektorie je nejpoužívanější trajektorií v k-prostoru [9]. Všechny řádky rastru jsou v k-prostoru vzájemně rovnoběžné, data jsou získávána řádek po řádku, viz obrázek 1.2. Data jsou čtena ve směru k_x souřadnic (prostorové kódování) a směr souřadnic k_y určuje fázové kódování. Typicky se kartézská trajektorie vyznačuje



Obr. 1.2: Kartézská trajektorie.

stejným rozestupem mezi řádky, takže je možné pro rekonstrukci obrazu použít uniformní FFT (rychlou Fourierovu transformaci).

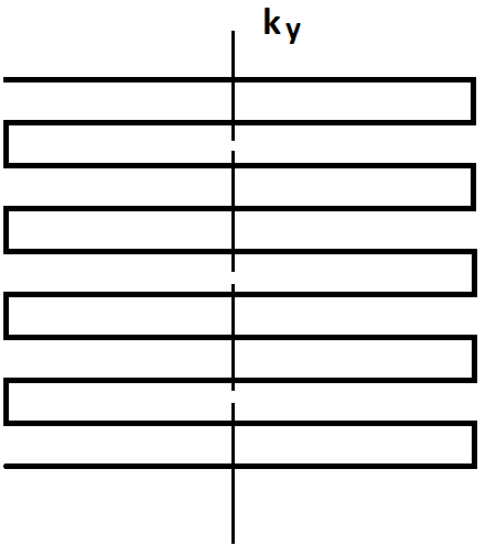
Trajektorie cikcak

Tato trajektorie prochází k-prostorem střídavě dopředným a zpětným směrem. Používá se buď s konstantním fázovým kódováním, kdy tato trajektorie netvoří pravoúhlou mřížku, takže je třeba k-prostor přepočítat do pravých úhlů, nebo s takzvaným blikajícím fázovým kódováním. V tomto případě jsou v trajektorii pravidelné kolmé přechody mezi směry dopředu a nazpět, takže je pravoúhlost dodržena, jak lze vidět na obrázku 1.3.

Radiální trajektorie

Radiální neboli také paprsčitá trajektorie sestává z paprsků vysílaných zevnitř středu k-prostoru kruhově směrem ven, viz obrázek 1.4. První vůbec použitá trajektorie v k-prostoru v MRI byla právě radiální [9].

Výhoda použití radiální trajektorie je dosažení vyššího odstupu signálu od šumu (SNR) než za použití ostatních trajektorií v k-prostoru. Z důvodu neuniformní vzorkovací hustoty radiální trajektorie je třeba delší skenovací doby, aby byl dodržen dostatečný vzorkovací kmitočet odpovídající Nyquistově poučce. V případě podvzorkování signálu se objevují artefakty v podobě pruhů. Pokud artefakty nezasahují do sledované oblasti, může nám použití metody podvzorkování signálu ušetřit skenovací čas za zachování stejného prostorového rozlišení, jakého bychom docílili použitím kartézské trajektorie. Z důvodu výše zmíněné neuniformní hustoty vzorků



Obr. 1.3: Trajektorie cikcak.

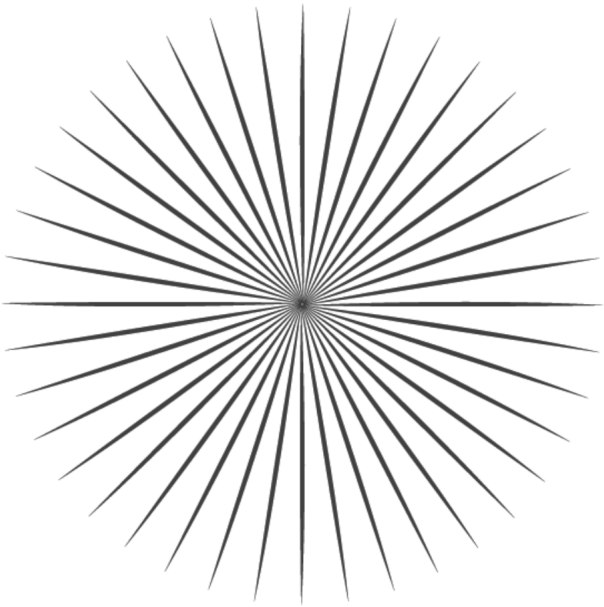
nestačí pro rekonstrukci obrazu klasická FFT, ale je třeba použít neuniformní rychlou Fourierovu transformace (NUFFT).

Spirálová trajektorie

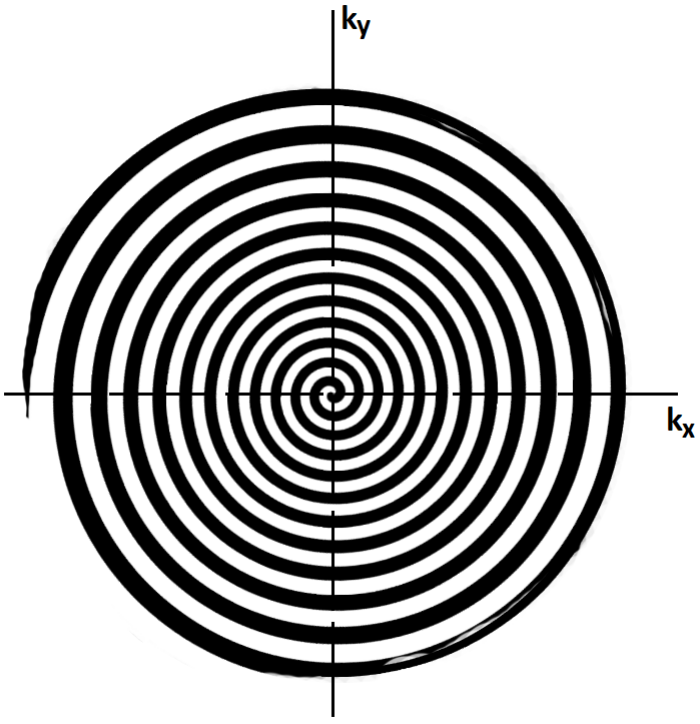
Spirálová trajektorie vznikla za účelem zkrácení skenovací doby a zároveň pokrytí k-prostoru efektivnějším způsobem, než lze získat použitím kartézské trajektorie. Má počátek ve středu k-prostoru, ze kterého se spirálovitě odvíjí směrem ven, jak je ilustrováno na obrázku 1.5.

Další trajektorie

Kromě výše uvedených, nejpoužívanějších trajektorií existuje množství dalších trajektorií. Např. stochastická (náhodné trajektorie), růžicovitá, kruhová nebo trajektorie hybridní (tj. kombinace dvou či více druhů trajektorií) [9].



Obr. 1.4: Radiální trajektorie.



Obr. 1.5: Spirálová trajektorie.

2 Rekonstrukce obrazu z MR

2.1 Statická rekonstrukce

Následující kapitola čerpá ze zdroje [3], [7], [10] a [11].

Získávání dat z magentické rezonance (MR) probíhá v dopředném směru, tedy z předlohy snímané tkáně x je Fourierovou transformací získán obraz v k-prostoru d, jak již bylo vysvětleno v kapitole 1.2. Můžeme tento proces zapsat rovnicí

$$d = \mathcal{F}x, \tag{2.1}$$

kde symbol $\mathcal{F}$ představuje operátor Fourierovy transformace. V praxi působí při získávání dat z magnetické rezonance také šum, který zanáší do výpočtu náhodnou proměnnou. V případě snímání pomocí více cívek, což je běžně používáno, je do výpočtu ještě zahrnuta citlivost cívek, tedy místo $\mathcal{F}x$ je rovnice psána $\mathcal{F}\mathcal{C}_c x$. Vzor x je násoben mapou citlivosti C_c a poté je na upraveném vzoru $\mathcal{C}_c(x)$ provedena Fourierova transformace. Celkově jsou operace tedy zapsány operátorem $\mathcal{F}\mathcal{C}_c x$.

2.1.1 Základní metody rekonstrukce

V případě rekonstrukce obrazu z magnetické rezonance jde o směr zpětný, kdy z obrazu v k-prostoru d se snažíme získat snímek předlohy x. Způsob, jakým lze vzor x vypočítat, záleží na použité trajektorii pro sběr dat. Pokud probíhá vzorkování v kartézské trajektorii, data jsou seskupena v rovnoměrné mřížce, a tedy existuje zpětný operátor Fourierovy transformace $\mathcal{F}^{-1}$, pomocí kterého lze předloha obrazu získat. Výpočet lze popsat rovnicí

$$x = \mathcal{F}^{-1}d. \tag{2.2}$$

Naopak pokud probíhá sběr vzorků ve směru trajektorie radiální, jedná se o neuniformní Fourierovu transformaci, která nemá inverzní funkci, pouze existuje sdružený operátor $\mathcal{F}^*$. Nexistuje-li inverze, nelze výše uvedený výpočet 2.2 použít. Předloha x není přímo vypočitatelná z obrazu d, jelikož řešení rovnice 2.1 může být více než jedno, a snažíme se tedy najít takové řešení, které je co nejpodobnější původnímu vzoru. Je k tomu možno použít metodu nejmenších čtverců nebo metodu minimální variace [10].

2.1.2 Regularizace metod rekonstrukce

Základní metody rekonstrukce nejsou příliš odolné vůči šumu a chybám při odhadu řešení rovnice rekonstrukce magnetické rezonance 2.1. Do výpočtu rekonstrukce je

tedy zavedena metoda regularizace pro kompenzaci těchto vlivů. Jednou z používaných metod regularizace je metoda totální variace (TV). Metoda využívá regularizačního parametru λ, který je třeba vhodně zvolit, aby vyvážil regularizační člen a datový člen, jež dohromady tvoří regularizovanou podobu metody rekonstrukce. Obecně vypadá matematický zápis regularizovaného problému pro různé metody rekonstrukce následovně [3]:

$$x_{\text{reg}} = \arg\min \left\{ \frac{1}{2} \sum_{c=1}^{N_c} \|d_c - \mathcal{F}\mathcal{C}_c x\|^2 + \lambda \cdot r(x) \right\}, \tag{2.3}$$

kde $\|d_c - \mathcal{F}\mathcal{C}_c x\|^2$ je datový člen a *r(x)* regularizační člen. Zkratka arg min značí argument minima, tj. hodnotu, ve které bude funkce rovna nule. Hledáme tedy takové řešení x_{reg}, které minimalizuje součet datového a regularizačního členu [10].

2.2 Dynamická rekonstrukce

Dynamická rekonstrukce umožňuje zachycení sledu snímků, je proto používaná pro získání informací o průběhu nějakého jevu. V lékařství je často spojená s injektáží kontrastní látky do krevního oběhu pacienta, jejíž šíření v čase poté snímáme za pomocí dynamické magnetické rezonance.

Do výše představené rovnice regularizované rekonstrukce 2.3 je nutno pro popis dynamické rekonstrukce přidat časové rámce. Ty odpovídají seskupení několika projekcí a jejich velikost určuje časové rozlišení. Problém dynamické rekonstrukce lze zapsat jako

$$x_{\text{reg}} = \arg\min \left\{ \frac{1}{2} \sum_{c=1}^{N_f} \sum_{f=1}^{N_c} \|d_{c,f} - \mathcal{F}_f \mathcal{C}_c x_f\|^2 + \lambda \cdot r(x) \right\}, \tag{2.4}$$

kde x_f představuje sled snímků a výraz $d_{c,f}$ odpovídá seskupeným datům do rámců v k-prostoru [3].

Pokud se zaměříme na snímání průtoku krve (perfuze) s kontrastní látkou, můžeme předpokládat, že lze sled snímků rozdělit do dvou částí. Je to nazýváno L+S modelem (low-rank plus sparse model), kde složka L představuje v čase pomalu se měnící částí obrazu mezi snímky a složka S dynamickou informaci v obrazu, která se v časové ose rychle mění [11].

2.2.1 Proximální algoritmy

Proximální algoritmy jsou používané v případě, kdy potřebujeme získat minimum funkce, jako v případě rekonstrukce z MR. Patří mezi iterační metody, to znamená, že z počátečního odhadu řešení počítáme posloupnost přibližných řešení, pokaždé poměrně jednoduchou iterací na základě předchozí iterace.

Jeden z těchto algoritmů je proximálně gradientní algoritmus (PG algoritmus), který se řadí mezi metody, které pracují střídavě v dopředném a zpětném směru. Správné nastavení délky kroku algoritmu zajistí konvergenci (sbíhavost) řešení. Zápis algoritmu pro řešení L+S modelu dynamické rekonstrukce magnetické rezonance je uveden v příloze A. V algoritmu je třeba nejdříve provést počáteční nastavení hodnot, a poté již běží iterace až do určeného maximálního počtu iterací.

3 Algoritmus SVD

3.1 Úvod

SVD je zkratka pro Singular Value Decomposition, neboli rozklad na singulární hodnoty. Jedná se o algebraickou transformaci matic, používanou ve zpracování obrazových signálů. V této kapitole byl použit zdroj [12].

3.2 SVD

SVD rozklad zajišťuje převedení maximálního množství energie do co nejméně koeficientů. Rozděluje matici do sady lineárně nezávislých součástí. SVD můžeme popsat pomocí následující rovnice [12]:

$$[\mathbf{X}] = [\mathbf{U}]\ [\mathbf{S}]\ [\mathbf{V}]^{\mathrm{T}}, \tag{3.1}$$

kde matici $\mathbf{X}$ o velikosti M×N (přičemž M≥N) rozkládáme na následující matice: ortogonální matici $\mathbf{U}$ o velikosti M×M, ortogonální matici $\mathbf{V}$ s rozměry N×N, která je v případě reálné matice transponovaná, v případě komplexní matice transponovaná konjugovaná[1], a diagonální matici $\mathbf{S}$ o rozměrech M×N. Jednotlivé matice jsou matematicky vyjádřeny v následujícím výrazu [12]:

$$S = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_n \end{bmatrix},\ \mathrm{U} = [\mathbf{u}_1, \mathbf{u}_2, \ldots \mathbf{u}_m],\ \mathrm{V} = [\mathbf{v}_1, \mathbf{v}_2, \ldots \mathbf{v}_n]. \tag{3.2}$$

Sloupce matice $\mathbf{U}$ nazýváme levými singulárními vektory matice $\mathbf{X}$, sloupce $\mathbf{V}$ pravými singulárními vektory $\mathbf{X}$. Prvky diagonály $\mathbf{S}$ odpovídají singulárním hodnotám matice $\mathbf{X}$. Každá singulární hodnota určuje jas obrazu a odpovídající dvojice singulárních vektorů specifikuje geometrii (topologii) obrazu.

3.3 Použití algoritmu

V rekonstrukci v MRI se vyskytují dva nejnáročnější výpočty, které je třeba aplikovat na velký objem dat a opakovaně je počítat znova pro každý krok rekonstrukce, a sice SVD a NUFFT [3]. První část praktického řešení práce se zaměřuje na algoritmus SVD pro účely implementace algoritmu a jeho zrychlení za pomoci paralelních

[1] Transponovaná konjugovaná matice je taková komplexní matice, na níž byla provedena transpozice a také komplexní sdružení.

výpočtů. V dalších částech práce následuje paralelizovaná implementace funkce NUFFT pomocí výpočtů na GPU.

4 Paralelizace výpočtů

4.1 Význam paralelních výpočtů v MRI

Paralelizace výpočtů znamená řešení vícero výpočtů souběžně oproti standardnímu sekvenčnímu způsobu výpočtů. Využívá se pro zvýšení výkonu, zefektivnění a zrychlení výpočetních operací. Hlavní problém v zobrazování magnetickou rezonancí je dlouhá skenovací doba, při které je třeba minimalizovat pohyb vyšetřovaných osob. Je tedy snaha dobu potřebnou ke sběru dat snížit, abychom se vyvarovali rizika nepřesnosti obrazu, zapříčiněné nechtěným pohybem vyšetřovaného. Paralelizací složitých úloh rekonstrukce obrazu získaného magnetickou rezonancí můžeme výpočty zefektivnit a potřebnou dobu snížit.

4.2 Princip

4.2.1 Typy paralelizace

Vytvoření paralelního programu musí předcházet určení, jaký typ paralelizace je pro náš účel nejvhodnější. Níže jsou popsané základní dva druhy paralelizace, přičemž studie čerpá ze zdroje [13].

Datový paralelismus

Jeden z předních problémů při vědeckých výpočtech spočívá v ohromných množstvích dat, které je třeba zpracovat. Sekvenční výpočty takovýchto objemů bývají časově velice náročné. Pokud lze pracovat současně s různými částmi dat více procesorovými jednotkami, jedná se o datový paralelismus. Tento typ paralelizace bývá taktéž označován zkratkou SPMB – Single Program Multiple Data, což vyjadřuje stav, kdy máme jeden program, ale více dat zpracovaných současně.

Úlohový paralelismus

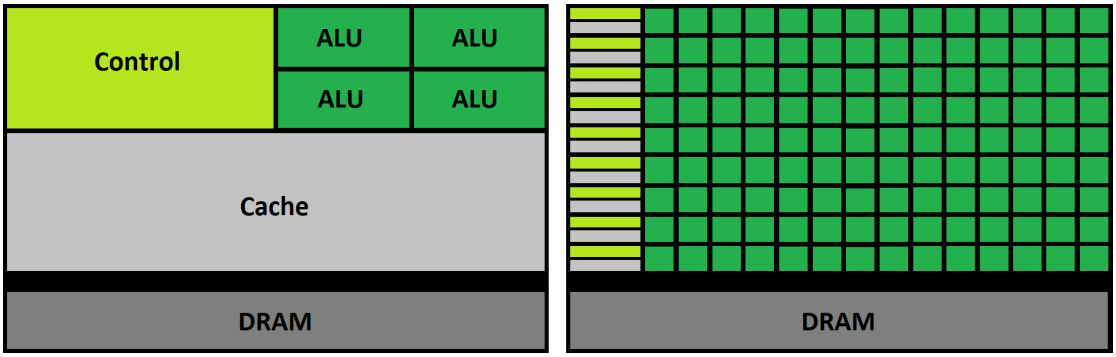
Dalším typickým příkladem, kdy je třeba paralelizace, je situace, kdy máme složitý výpočetní problém. Při použití sekvenčních výpočtů trvá vyřešení dlouhou dobu. Jsme-li ale schopni problém rozdělit do více jednodušších podproblémů, můžou probíhat současně různé výpočty na různých datech. Je třeba ale zajistit výměnu dat mezi podproblémy a jejich synchronizaci. Tento druh paralelizace se nazývá úlohový paralelismus a lze ho také najít pod zkratkou MPMD – Multiple Program Multiple Data, tedy více programů zpracovává více dat současně. Je také možné, aby byly paralelismy zkombinovány, neboli rozdělené podproblémy zpracovávají různými

procesorovými jednotkami paralelně různé části dat podle paralelizace typu SPMB. Efektivita úlohového paralelismu se odvíjí od toho, jak podobné výpočetní nároky probíhají současně. Pokud jsou v tomto směru ekvivalentní, jsme schopni ušetřit významné množství času, pokud však mají různé výpočetní nároky, náročnější úloha představuje překážku, do jejíhož dokončení ostatní výpočty stojí, a tím efektivnost paralelizace klesá.

4.2.2 Rozdíl mezi GPU a CPU

Abychom byli schopni vysvětlit, jak funguje paralelizace úloh pomocí výpočtů na grafickém procesoru grafické karty, je třeba představit rozdíly mezi procesorem počítače a procesorem grafické karty.

Z hlediska architektury je zásadní rozdíl mezi CPU (centrální procesorovou jednotkou počítače) a grafickým procesorem, neboli zkráceně GPU (z anglického Graphics Processing Unit), viz na obrázku 4.1. CPU je složena z několika málo jader (aritmeticko-logických jednotek – ALU), ve kterých se provádějí aritmetické a logické operace, zato z obsáhlé mezipaměti (anglicky Cache). Je tak schopna zpracovat jen malý počet vláken procesu zároveň. Oproti tomu grafický procesor obsahuje stovky jader, které umožňují zpracování tisíce vláken současně [14].



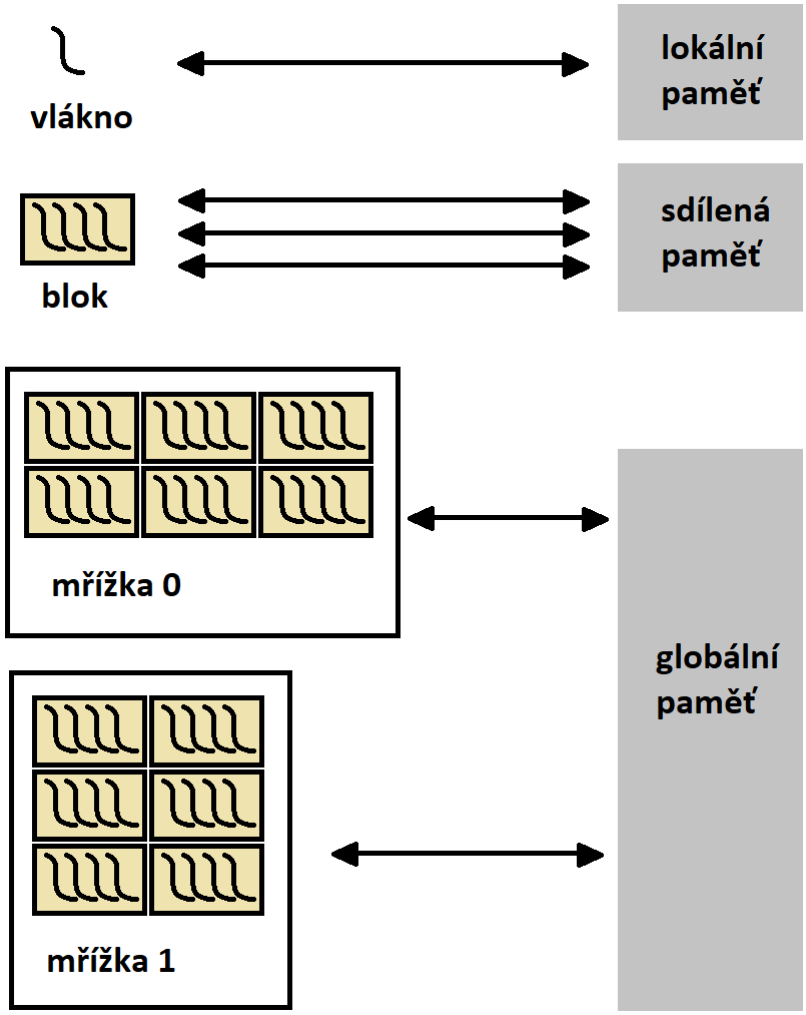
Obr. 4.1: Srovnání architektury CPU (vlevo) a GPU (vpravo).

Takováto možnost paralelizace znamená až stonásobné zrychlení výpočtů oproti CPU. Navíc GPU dosahuje větší efektivity využití prostředků, a tedy větší hospodárnosti [14].

Oproti centrální procesorové jednotce však není věnována přednost v architektuře GPU kontrolní logice (na obrázku 4.1 je označena anglicky Control) a mezipaměti (Cache). GPU je tedy nejvhodnější použít pro problémy, jež lze paralelizovat, protože je při paralelních výpočtech vysoký poměr aritmetických operací ku operacím s pamětí a jelikož při paralelních výpočtech dochází k spuštění jednoho programu pro velké množství dat, jsou nižší nároky na kontrolní logiku procesoru [16].

4.2.3 Hierarchie pamětí GPU NVIDIA

Tato část práce čerpá z dokumentace NVIDIA [16] a zabývá se abstrakcí paměti v grafických procesorech. Paměť je tak rozdělena na globální, lokální, sdílenou, konstantní a texturovou paměť. Hierarchie paměti je zobrazena na obrázku 4.2.



Obr. 4.2: Hierarchický model paměti GPU NVIDIA.

Globální paměť

Všechna vlákna mají přístup ke stejné globální paměti. Globální paměť je uložena v paměti DRAM (dynamické paměti s přímým přístupem – Dynamic Random Access Memory) grafického procesoru a umožňuje čtení/zápis po částech o maximální délce 128 bajtů. Globální paměť má slabinu ve zpoždění, lze je ale skrýt načítáním následujících dat současně s výpočty s daty aktuálními při dostatečně intenzivní zátěži aritmetickými operacemi. Pro zvýšení propustnosti globální paměti je třeba

organizace paměti tak, aby navazující data spolu sousedila. Čím více jsou data roztýlena, tím nižší propustnost paměť dosahuje. Tato optimalizace paměti je v případě globální paměti nejpodstatnější, protože kvůli obecně nejnižší propustnosti má globální paměť v případě nedostatečné organizace zásadní účinek na celkový výkon procesoru.

Lokální paměť

Každé vlákno má vlastní lokální paměť. Je fyzicky částí globální paměti a je použita v případě, že nedostačuje počet registrů či data přesahují velikost registru. Vlákna mají k dispozici nanejvýš 255 32bitových registrů a ukládají do nich data jen po dobu existence vlákna. Jelikož je lokální paměť součástí globální paměti, na rozdíl od obecně nejrychlejšího přenosu při práci s registry dosahuje lokální paměťový prostor stejně nízkých rychlostí přenosu jako globální paměť. Je na rozdíl od ní ale organizován, aby navazující 32bitová slova četla vlákna se sousedními identifikátory (ID), tím je dosaženo výše zmiňované optimalizace.

Sdílená paměť

Každý blok má sdílenou paměť viditelnou pro všechna vlákna příslušného bloku. Sdílená paměť leží na čipu procesoru, tudíž umožňuje přenos výrazně rychlejší a s menším zpožděním než globální paměť. Dosahuje toho pomocí rozdělení sdílené paměti na stejně velké moduly paměti nazývané banky, ke kterým je možno přistupovat současně. Pokud tedy přijde n žádostí o přístup do n paměťových bank, mohou být zpracovány zároveň, a tedy se zvýší rychlost oproti práci jednoho modulu v jeden moment n-krát. Pokud však vzniknou dvě žádosti na přístup do stejné paměťové banky, nastává konflikt a přístup musí být přidělen popořadě (sériově). Hardware GPU rozdělí konfliktní žádosti do tolika (předpokládejme n) pod-úkolů, které konflikt nevytvářejí, kolik je potřeba, čímž se sníží rychlost zpracování n-krát. Abychom tedy zabránili snížení rychlosti, je třeba plánovat žádosti o přístup do paměti, abychom konflikty minimalizovali.

Ostatní paměti

Kromě výše uvedených ještě existují paměti pouze ke čtení přístupné všemi vlákny, a sice konstantní a texturový paměťový prostor.

Konstantní paměťový prostor je koncipován na přístup více vláken zároveň ke stejnému obsahu a rychlost je úměrně nižší počtu současně přistupujících vláken. Má také mezipaměť na čipu, takže výsledná rychlost přístupu je rychlejší než u globální paměti. Obsahem bývají data neměnná v průběhu funkce jádra a vlákna mají pouze možnost čtení z paměti.

Texturový paměťový prostor je podobný konstantní paměti. Umožňuje také pouze režim čtení, ale je optimalizován pro uložení vícerozměrných dat. Má mezipaměť v čipu, do které textury ukládá, takže je třeba do paměťového prostoru na DRAM přistupovat pouze, pokud žádosti přesáhnou obsah mezipaměti.

4.3 CUDA

4.3.1 Úvod

Aby byla paralelizace úloh v GPU spuštěna, je třeba použít rozhraní, které komunikaci s grafickým procesorem umožňuje. K tomuto účelu představila v roce 2006 společnost NVIDIA programovací model a vývojovou platformu nazvanou CUDA – Compute Unified Device Architecture.

4.3.2 Popis

Tato kapitola čerpá ze zdroje [17]. CUDA obsahuje programovací prostředí vyvinuté společností NVIDIA pro paralelní výpočty na NVIDIA grafických procesorech grafické karty, algoritmů naprogramovaných v jazyce C. CUDA umožňuje vývojářům přístup k sadě instrukcí a paměti paralelních výpočetních prvků NVIDIA grafických procesorů.

Grafické procesory obsahují architekturu, která umožňuje provádění množství časově souběžných vláken procesů zároveň na rozdíl od principu spouštění jednoho či několika málo vláken současně, jež umožňuje CPU. Každou souběžnou (paralelní) funkci na hostitelském počítači nazýváme funkcí jádra (anglicky Kernel Function) a je spuštěna na mřížce v GPU.

Každá mřížka obsahuje skupinu bloků seskupených do podoby dvourozměrného pole. Bloky jsou pak seřazeny do třírozměrného pole (v celkovém počtu až 512) vláken. Po spuštění jádra nelze měnit jeho rozměry. Jádro je definováno dvěma parametry, a sice počtem bloků v mřížce a počtem vláken v každém bloku. Každé vlákno má vlastní unikátní index v rámci bloku a spojením indexu bloku a vlákna lze získat jedinečný index pro každé vlákno mřížky.

Rozlišujeme dvě spolupracující entity – hostitele (anglicky Host), neboli počítač (zkráceně PC – Personal Computer), a na druhé straně grafický procesor. Funkce na GPU pracují s pamětí, software proto poskytuje funkce na alokování, dealokování a zkopírování paměti grafického procesoru, stejně jako přesunutí dat mezi hostitelem a GPU.

4.3.3 Pojmy v prostředí CUDA

Funkce jádra je definována použitím deklarace `_global_` a dvěma parametry v závorkách `kernel<<<...>>>` – počtem bloků v mřížce a počtem vláken v bloku. Jedinečnou identifikaci každého vlákna zajišťuje tří-složkový vektor `ThreadIdx`, popisující funkci jádra a blok, pod které vlákno spadá, v prvních dvou složkách a ve třetí odlišuje vlákno v rámci daného bloku.

Paměť grafického procesoru můžeme alokovat dvěma způsoby, buď jako lineární paměť, kdy může procesor používat sekvenční adresování bez potřeby segmentace paměti, čehož je obvykle využíváno ve vývoji v nižších programovacích jazycích, nebo ve formě pole s vlastním CUDA rozložením paměti. Druhá zmíněná možnost je optimalizovaná pro práci s texturami a je možné k paměti přistupovat jen ve formě paměti texturové.

Lineární paměť lze alokovat voláním funkce `cudaMalloc()` a poté opět uvolnit použitím funkce `cudaFree()`. Výměna dat mezi pamětí PC a GPU je provedena voláním funkce `cudaMemcpy()`. Sdílená paměť je deklarována za pomocí specifikace `_shared_`.

Souběžné operace probíhají ve formě datových toků, označovaných jako `stream`. Datový tok sestává z řady příkazů, které se provádějí popořadě. Různé toky mohou zato provést operace současně.

Výpočetní možnosti (anglicky **Compute capability** grafického procesu popisují vlastnosti zařízení a jsou reprezentovány číslem verze. Určují, jaké části softwaru jsou podporovány hardwarem grafického procesoru, a tato informace je předána aplikacím. Verzi výpočetních možností je třeba nezaměňovat s verzí softwarového prostředí CUDA, i když spolu souvisí. Od verze CUDA 7.0 nejsou podporovány architektury s verzí výpočetních možností nižší než 3.0.

Funkce jádra v programovém prostředí CUDA musí být zkompilovány do binárního kódu pomocí **NVCC** – NVIDIA CUDA kompilátoru, který je spustitelný na grafickém procesoru.

5 Praktická část

5.1 Hardware

5.1.1 GPU

Ke zpracování dat byla k převážné většině měření použita primární grafická karta NVIDIA TITAN Xp s architekturou NVIDIA označovanou kódovým názvem Pascal a výpočetními možnostmi (compute capability – viz podkapitola 4.3.3) verze 6.1. V následující tabulce (5.1) je přehled hlavních technických informací o grafickém procesoru grafické karty NVIDIA TITAN Xp.

Tab. 5.1: Výběr z přehledu technických informací GPU primární grafické karty NVIDIA TITAN Xp [16].

Maximální počet spuštěných funkcí jádra současně	32
Počet ALU jednotek (jader)	128
Velikost paměti	12 GB
Propustnost paměti	547,7 GB/s

Sekundární grafická karta s architekturou NVIDIA označovanou kódovým názvem Maxwell výpočetními možnostmi verze 5.0. byla použita v závěrečné části práce 5.9, kdy byla naměřena srovnání rychlostí optimalizované funkce NUFFT na grafickém akcelerátoru pomocí knihovny JCuda a technologie CUDA.

Tab. 5.2: Výběr z přehledu technických informací GPU sekundární grafické karty NVIDIA GTX 750 Ti [16].

Maximální počet spuštěných funkcí jádra současně	32
Počet ALU jednotek (jader)	128
Velikost paměti	2 GB
Propustnost paměti	86,4 GB/s

5.1.2 CPU

Pro účel srovnání byly výpočty prováděny na procesoru počítače. Testování proběhlo na CPU Intel® Core™ i7-2600 s taktovací frekvencí 3.4 GHz. V tabulce níže (5.3) je uvedený přehled technických specifikací paměti RAM (paměť s přímým přístupem – Random Access Memory) a procesoru na PC.

Tab. 5.3: Výběr z přehledu technických informací CPU a RAM.

Počet ALU jednotek (jader)	4
Velikost paměti RAM	16 GB
Propustnost paměti RAM	10,67 GB/s

5.2 Software

5.2.1 Vývojové prostředí

Rekonstrukční model

Zadaný rekonstrukční model spolu se vstupními daty pro MRI byl zprovozněn v programu MATLAB 2017b od společnosti MathWorks.

Implementace v jazyce Java

Volání instance MATLABu, zpracování dat a implementace algoritmu pro paralelizaci byly provedeny v jazyce Java v open-source vývojovém prostředí Eclipse Photon (vyvíjeném společností Eclipse Foundation) a paralelizace bylo dosaženo za použití vývojového rozhraní CUDA (spolu se souborem nástrojů JCuda) pro přístup a komunikaci s grafickým procesorem.

5.2.2 JCuda

Následující text popisuje soubor nástrojů pro vývoj kódu pro GPU v jazyce Java nazvaný JCuda. Podkapitola čerpá z oficiální dokumentace [18].

JCuda poskytuje dvě různá programovací rozhraní aplikací (zkráceně API [1]), a sice API ovladače a API pro běh programu. Jsou si vzájemně podobné v základních operacích (např. práce s pamětí), ale hlavní rozdíl mezi nim je ve způsobu, jakým jsou jádra spravována a funkce jádra prováděna. Kód napsaný v jazyce Java nelze přímo kompilovat pomocí NVCC jako v případě prostředí CUDA, kde jsou instrukce psány v jazyce C. Pokud chceme vytvářet vlastní funkce jádra, je třeba použít API ovladače, pokud je vytvářet nepotřebujeme, ale pouze nám stačí využití CUDA knihoven pro běh programu, můžeme použít API pro běh programu, které umožňuje spolupráci s knihovnami CUDA.

[1] Application Programming Interface

5.2.3 gpuArray

Objekt, jenž je nazýván v programovacím prostředí MATLAB gpuArray [19], představuje datové pole, uložené v paměti grafického procesoru grafické karty. Je možné objekt gpuArray použít pro paralelní výpočty přímo v MATLABu za využití funkcí, které tento datový typ podporují, anebo také ve funkcích jádra CUDA, jež spustí výpočty na grafickém procesoru.

Kromě datového pole je gpuArray také funkce, která tento datový typ vytvoří, neboli jeho konstruktor. Příkazem `G = gpuArray(X)` dojde ke zkopírování vstupních dat X do paměti grafického procesoru, a k datům lze v MATLABu přistupovat ve formě objektu G typu gpuArray. Pokud potřebujeme data přesunout z grafického procesoru grafické karty zpátky do paměti počítače, slouží k tomuto účelu příkaz `Y = gather(G)`. Přesuny mezi pamětí GPU a CPU jsou prostředkově velice náročné a zvyšují celkový čas výpočetního úkonu. Je tedy obecně doporučeno snížit výskyt těchto operací v kódu na minimum, a použít je jen v místech, kde není práce s datovým polem, uloženém v paměti GPU možná.

V případě, že potřebujeme přistupovat přímo ke grafickému procesoru v prostředí MATLAB, lze se odkazovat na objekt `gpuDevice`, reprezentující používaný GPU. Lze zjistit stav a vlastnosti procesoru, například zbývající volná paměť, a pomocí příkazu `reset` je možné uvolnit paměť grafického procesoru.

5.3 Vstupní data MRI

Reálná vstupní data MRI byla dodána spolu s rekonstrukčním modelem vedoucím práce. Jedná se o dvourozměrné snímky mozku laboratorní myši snímané v radiální trajektorii periodicky v časové ose.

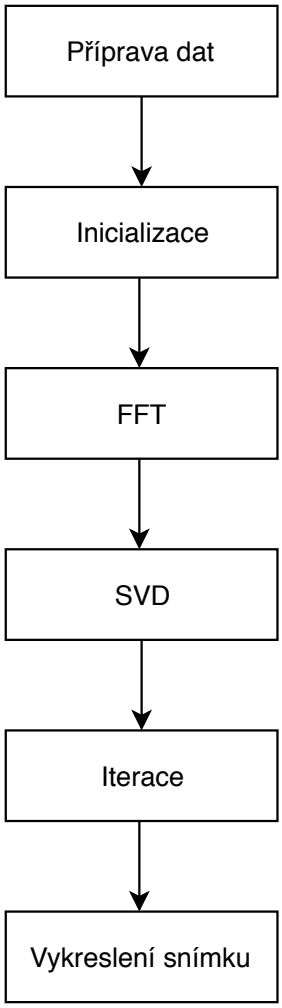
Data jsou načtena v programu MATLAB a uložena ve formě čtyřrozměrných polí datového typu `Complex`. Rozměry jednotlivých dimenzí jsou dány nastavením sběru dat při snímkování MR. Pro snímání byly použity 4 cívky, každá cívka zobrazuje obraz o velikosti 128 bodů. Projekcí na snímek je 21 a snímků bylo sesbíráno skenováním celkově 50 000.

Předzpracováním dat v rekonstrukčním modelu jsou data převedena do 2D pole komplexních čísel o rozměrech 16384×2373. Toto pole, převedené do 1D vektoru, slouží jako vstupní proměnná do implementace algoritmu SVD za pomocí nástroje JCuda, zpracované v podkapitole 5.6.1.

Výše popsaná vstupní čtyřrozměrná data, nahraná v MATLABu, jsou spolu s pomocnými proměnnými, které do dalších náročných úloh v rekonstrukci MRI vstupují, uložena v programu v jazyce Java, a následně převedena do 1D vektoru.

5.4 Popis rekonstrukčního modelu v MATLABu

Zadaný rekonstrukční model v MATLABu sestává z několika fází, které na sebe navazují. Model postupuje od přípravy dat až po vykreslení rekonstruovaného snímku magnetické rezonance. Ve schématu 5.1 je ilustrováno, jak fáze programu rekonstrukčního modelu navazují. V následujících podkapitolách budou představeny podrobněji. Od fáze inicializace po iteraci včetně jsou délky výpočtu měřené v každé z fází spolu s výpisem názvu probíhající fáze modelu (viz příklad výpisu výsledků v příloze B.1).



Obr. 5.1: Fáze programu rekonstrukce MRI v MATLABu.

5.4.1 Příprava dat

V první fázi modelu probíhá načtení reálných vstupních dat ze skeneru, načtení citlivosti cívek použitých pro sběr dat a na jejich základě vytvoření operátoru pro

výpočty, ve kterých citlivosti cívek figurují.

5.4.2 Inicializace

V této fázi modelu je proveden počáteční (hrubý) odhad rekonstruovaného obrazu, neboli podle výše představeného proximálně gradientního algoritmu (v algoritmické podobě uvedeného v příloze A) je vypočítán obraz M ze vstupních dat d pomocí sdružených operátorů Fourierovy transformace a citlivosti cívek. Výpočet lze zapsat rovnicí:

$$M = \mathcal{F}^*\mathcal{C}^*(d) \tag{5.1}$$

Jsou zde postupně volány funkce pro výpočty za pomocí jednotlivých operátorů, jejichž průběh je ilustrován na následujícím schématu 5.2. Všechny funkce, zobrazené ve schématu, jsou součástí balíčku nástrojů ESPIRiT.

V zobrazené kaskádě operací se vyskytují dvě stejnojmenné funkce `mtimes.m`, jedna provádí funkci `nufft_adj.m`, která odpovídá operaci sdruženým operátorem Fourierovy transformace, a druhá aplikuje ESPIRiT operátor citlivosti, v kódu označovaný také jako Eigen-Vecs operátor. V této fázi rekonstrukce je použit operátor citlivosti sdružený, jak je naznačeno již v rovnici 5.1.

5.4.3 FFT

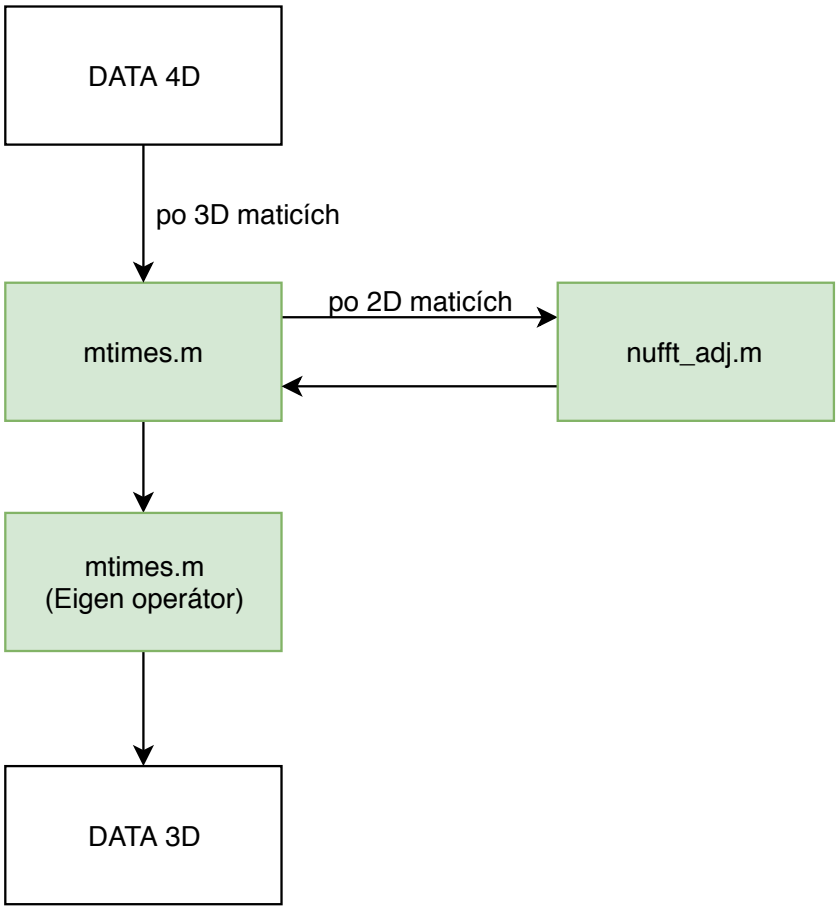
Touto fází začíná samotná rekonstrukce obrazu. Probíhá zde gradientní krok proximálně gradientního algoritmu, kdy na hrubý odhad rekonstruovaného obrazu M aplikujeme nejprve operátory Fourierovy transformace a citlivosti cívek, od transformovaných dat odečteme původní data d, a na jejich výsledku jsou poté provedeny operace pomocí sdružených operátorů. Jedná se o následující část algoritmu (A):

$$M^{(n+1)} = \mathcal{C}^*\mathcal{F}^*(\mathcal{F}\mathcal{C}(M^{(n)}) - d). \tag{5.2}$$

Jsou zde opět postupně volány funkce z balíčku nástrojů ESPIRiT, nejprve v podobě nesdružených operátorů, jak je zaznačeno na obrázku 5.3. V rámci těchto výpočtů je volaná funkce NUFFT, v rekonstrukčním modelu ve formě funkce `nufft.m`. Následně je na výsledku provedena stejná kaskáda funkcí jako ve fázi inicializace s pomocí sdružených operátorů, viz 5.2.

5.4.4 SVD

V této části programu probíhá algoritmus rozkladu na singulární hodnoty (SVD), již představený v kapitole 3.



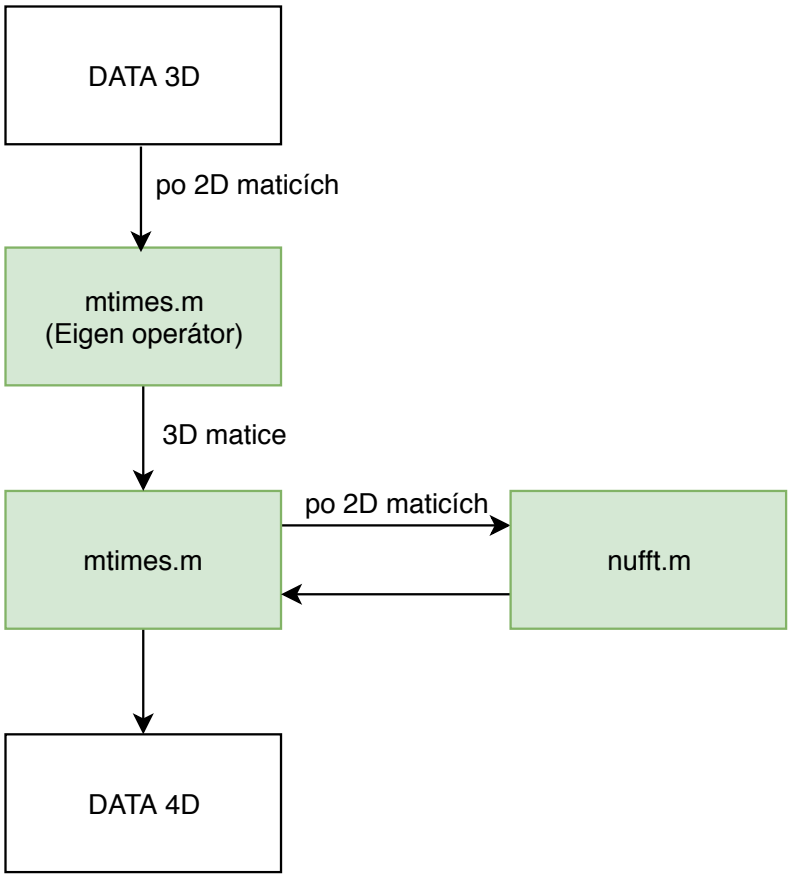
Obr. 5.2: Volání funkcí ve fázi inicializace rekonstrukčního modelu.

5.4.5 Iterace

V programu probíhají iterace až do stopovacího kritéria `maxit`, neboli parametr, udávající maximální počet iterací, které proběhnou. Měřená doba výpočtů v této fázi modelu zahrnuje aktualizaci L a S složek, jejich součet, čímž získáme obraz M, a posun počítadla iterací na další ($n = n + 1$), je-li maximální počet iterací více než jedna.

5.5 Blokové schéma řešení s paralelními výpočty pomocí nástroje JCuda

Níže je na obrázku 5.4 zobrazení vypracovaného provedení zpracování dat pro spuštění paralelních výpočtů na grafickém procesoru.
Schéma ukazuje proces zpracování, úpravu a přenos dat mezi vývojovými prostředími a procesory. Vstupní data jsou nahrána a předzpracována v MATLABu. V Javě je zavolána instance MATLABu a vstupní data jsou převedena z vícerozměrných

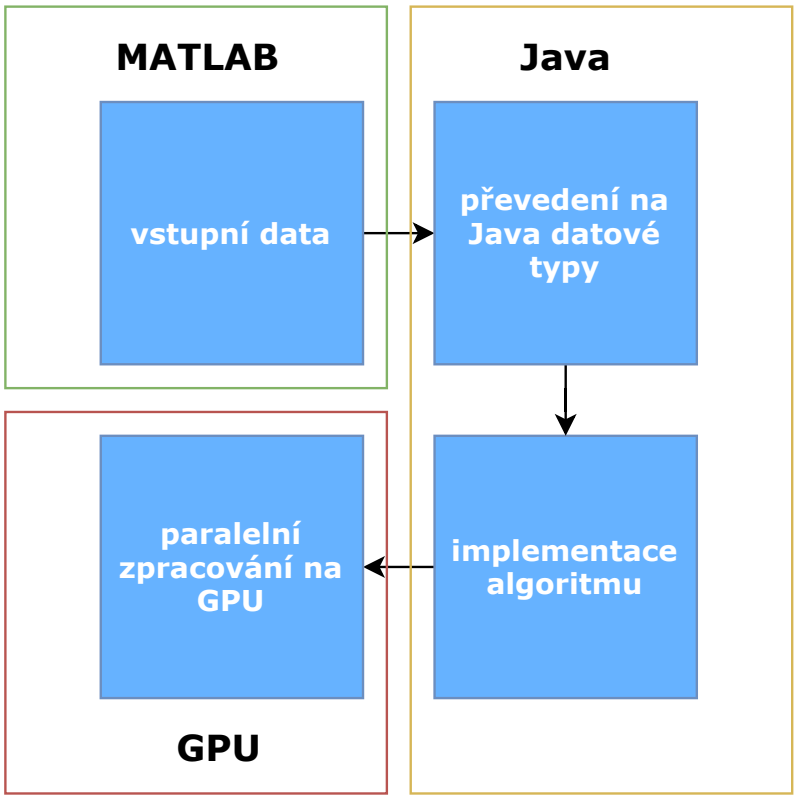


Obr. 5.3: Volání funkcí ve fázi FFT rekonstrukčního modelu - první část gradientního kroku.

matic do 1D polí `float[]` a uložena do datového typu `LinkedHashMap`. Jsou takto připravena na vstup grafického procesoru. Následují funkce, které data odesílají za pomoci volání knihoven platformy CUDA na paralelní zpracování do GPU, a zpracovávají vypočítané výstupy výpočtů.

5.6 Testování algoritmu SVD

Aby bylo možné zhodnotit výkonnost paralelního zpracování výpočtu SVD, je důležité srovnat rychlost výpočtu s implementací algoritmu, provádějící výpočty na CPU. V první části testování je třeba představit vypracované obdoby funkce SVD a původní algoritmus a následně vyhodnotit výkonnost výpočtů za použití různého typu procesoru na stejných vstupních datech.



Obr. 5.4: Blokové schéma zpracování dat.

5.6.1 Použité implementace algoritmu

Pro srovnání implementace algoritmu v MATLABu pracující s GPU byl použit výpočet SVD, který je součástí zadaného rekonstrukčního modelu v programu MATLAB, jenž bylo cílem zefektivnit.

SVD rekonstrukčního modelu

V dodaném rekonstrukčním modelu je pro výpočet SVD použita zabudovaná funkce programu MATLAB SVD v definici `[U,S,V] = SVD(X)`, kde X jsou vstupní data a U, S a V jsou výstupní rozložené matice, viz popis algoritmu v sekci 3.2.

SVD implementace v MATLABu

Pro srovnání byla použita SVD implementace v programu MATLAB voláním stejné funkce jako v předchozím případě, ale za použití objektu typu gpuArray. Pro testování byl použit kód vycházející ze zdroje [20], obsahující měření doby výpočtu.

SVD implementace v Javě

Data zpracována výše uvedeným způsobem (viz 5.5) jsou přivedena do vypracované metody `getSVD`. Pro výpočet SVD na GPU využívá vytvořená implementace algoritmu knihovnu JCusolver, čerpající z knihovny rozhraní CUDA s názvem cuSOLVER, jejíž součástí je metoda `cusolverDnCgesvd`. Program pracuje s upravenými daty ve formě 1D pole, a je třeba na vstupu metody alokovat také paměť pro výstupní matice U, S a V.

5.6.2 Naměřené hodnoty

Délka provedení jednotlivých implementací byla opakovaně změřena pro srovnání časové efektivnosti výpočtů. Získané časové údaje jsou uvedeny v tabulce 5.4 v sekundách.

Tab. 5.4: Srovnání rychlosti provedení algoritmu SVD.

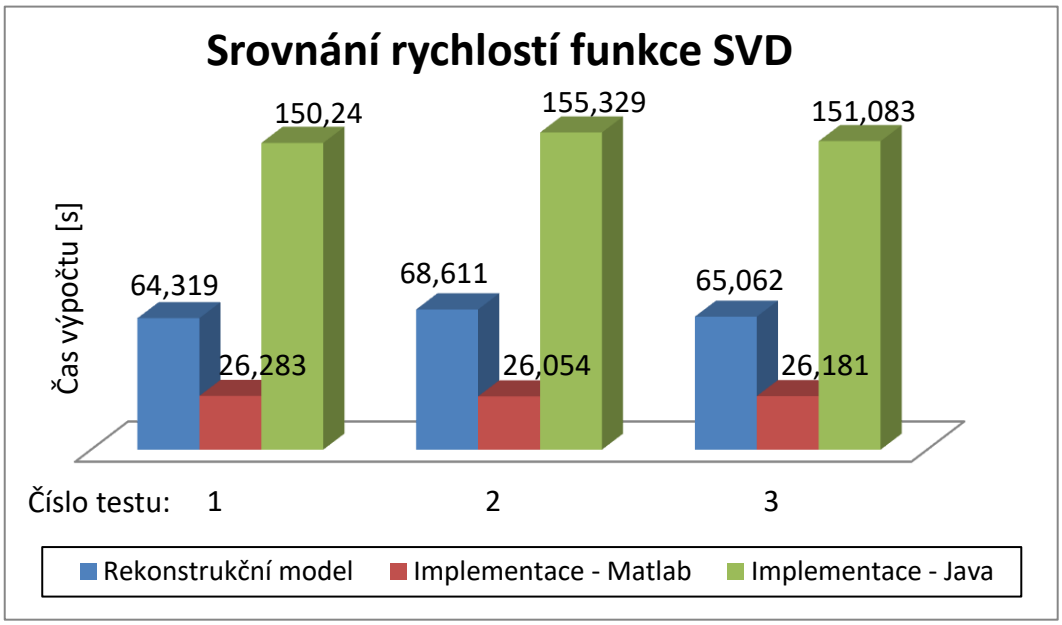
Verze algoritmu	Čas 1 [s]	Čas 2 [s]	Čas 3 [s]
Rekonstrukční model	64,319	68,611	65,062
MATLAB s použitím GPU	26,283	26,054	26,181
Java s použitím GPU	150,24	155,329	151,083

Z naměřených hodnot je zjevné, že implementace algoritmu v MATLABu s paralelními výpočty na GPU je časově nejúspornější, délka výpočtu byla přibližně 26 s. Naopak vypracovaná implementace v Javě, využívající knihovnu JCusolver, byla nejméně časově úsporná, výpočet trval přes 150 sekund.

5.6.3 Vyhodnocení výsledků

Srovnání naměřených hodnot je graficky zobrazeno na obrázku 5.5. Původní výpočet SVD v rámci dodaného rekonstrukčního modelu trval průměrně 66 sekund. Paralelizací výpočtů v programu MATLAB došlo ke zrychlení úlohy přibližně 2,5krát, délka výpočtu byla průměrně rovna 26,2 s. Paralelní zpracování oproti původnímu sekvenčnímu zpracování dat tedy mělo za následek významnou časovou úsporu.

Implementace úlohy v jazyce Java, využívající taktéž paralelních výpočtů na GPU, se ukázala jako pomalejší než zadaný rekonstrukční model v MATLABu. Výpočet tímto řešením trval průměrně 152 sekund. Nízká časová úspora řešení je dána využitím metody knihovny cuSOLVER pro výpočet SVD `cusolverDnCgesvd`, jejíž vývoj nebyl značnou dobu modernizován [21], a tak ve srovnání s výpočty na procesoru počítače nedosahuje takové efektivity. Největší časové ztráty jsou způsobeny



Obr. 5.5: Grafické srovnání rychlosti výpočtu algoritmu SVD

častým přenosem malých objemů dat mezi pamětí počítače a grafickým procesorem. Pro řešení optimalizace dalších náročných úloh v rekonstrukci magnetické rezonance bylo na základě získaných výsledků rozhodnuto nepokračovat tímto způsobem v řešení.

5.7 Implementace modelu v MATLABu s pomocí funkce gpuArray

5.7.1 Popis implementace

Řešení pomocí gpuArray bylo vypracováno v MATLABu na základě původního rekonstrukčního modelu. Aby bylo možno provádět paralelní výpočty efektivně a tak dosáhnout časové úspory, bylo třeba přepracovat model do podoby, která využívá datového typu gpuArray pro vybrané náročné úlohy, zároveň ale používané funkce mohou být volány při ostatních funkcionalitách modelu za použití původních datových typů (typicky `double` nebo `Complex double`). Dále bylo třeba používat nástroj gpuArray takovým způsobem, aby byly minimalizované přenosy dat mezi pamětí počítače a pamětí grafického procesoru, které jsou příčinou největších časových ztrát v paralalizaci s pomocí grafického akcelerátoru.

První problém byl ošetřen zavedením podmínek v kódu funkcí, které je potřeba volat při práci na CPU i na GPU. Tyto podmínky kontrolují vstupní datový typ a následně na základě jejích vyhodnocení k datům přistupují odpovídajícím způsobem.

Druhý problém byl vyřešen prvotním nahráním potřebných dat do paměti grafického procesoru, a následné zpracování dat již v rámci GPU upravenými funkcemi, které pracují s poli gpuArray. Přesuny výsledků zpět do paměti počítače tak v průběhu výpočetně náročných částí programu neprobíhají. Data jsou v rámci fáze iterace poté přesunuta zpět do paměti počítače pro výpočet vykreslení výsledků.

Srovnáme-li výpočty v modelu se zápisem proximálně gradientního algoritmu v příloze A, probíhají na grafickém procesoru výpočty až do aktualizace složky L. Řádek algoritmu

$$S^{(n+1)} = \text{prox}_{t\lambda_S r_S}(M^{(n+1)} - L^{(n+1)}), \tag{5.3}$$

neboli aktualizace složky S probíhá již na procesoru počítače. Výpočtem totální variace složky S na grafickém procesoru s pomocí gpuArray nebyla získána žádná časová úspora ani ztráta, bylo tedy vhodnější uvolnit paměť grafického procesoru, který už není pro výpočty potřeba.

5.7.2 Naměřené hodnoty

Délka provedení jednotlivých fází rekonstrukčního modelu byla v MATLABu opakovaně změřena pro srovnání časové efektivnosti výpočtů mezi původním modelem a jeho paralelizovanou implementací za pomoci gpuArray. Výsledky měření jsou pro jednotlivé fáze rekonstrukce uvedeny pro větší přehlednost v tabulkách v příloze. Tabulka výsledků pro fázi inicializace B.1, pro fázi FFT B.2, pro fázi SVD B.3 a pro fázi iterace B.4 uvádí naměřené délky výpočtů dané fáze získané ze tří měření výpočtů na procesoru počítače a ze tří měření výpočtů na grafickém procesoru pomocí gpuArray.

Naměřené hodnoty byly poté pro každou fázi zprůměrovány a uvedeny ve společné tabulce 5.5 pro srovnání časové úspory, získané optimalizací modelu s pomocí grafického akcelerátoru. Pro přehlednost jsou srovnány průměrné hodnoty také v grafu B.1, uvedeném v příloze.

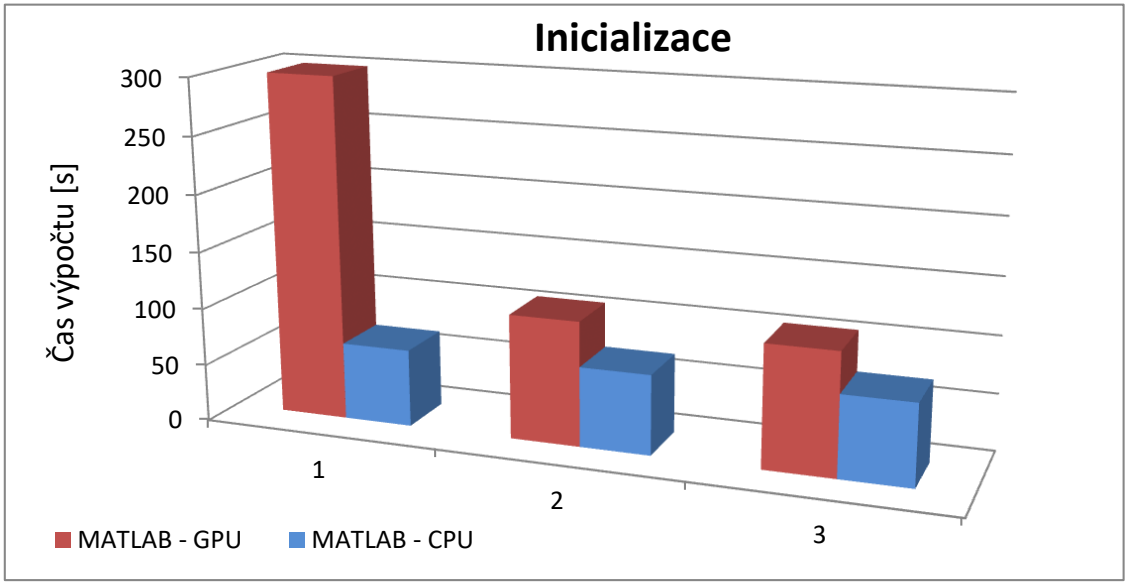
Tab. 5.5: Srovnání průměrné doby výpočtů jednotlivých fází rekonstrukčního modelu v MATLABu mezi původním programem a optimalizovanou implementací na GPU.

Fáze rekonstrukce	Doba výpočtu (CPU) [s]	Doba výpočtu (GPU) [s]
Inicializace	69,587	174,431
FFT	88,140	77,899
SVD	65,147	42,865
Iterace	96,279	88,644

Z tabulky 5.5 je patrné, že všechny fáze rekonstrukce kromě inicializace proběhly průměrně rychleji za vyžití gpuArray než v původním rekonstrukčním modelu.

5.7.3 Vyhodnocení výsledků

Srovnání naměřených časů pro fázi iterace je graficky zobrazeno na obrázku 5.6. Z grafu lze vyčíst, že ačkoliv průměrná doba výpočtů na GPU je více než dvojnásobná oproti výpočtům na CPU, prvotní výpočet, který trval 307,84 sekund, se hodně vychýlil naměřeným časem oproti dalším výpočtům, které trvaly přibližně 106 s a 108 s. V dalších měřeních už je čas trvání výpočtů jen zhruba o polovinu delší než při zpracování na CPU, které trvalo průměrně 69,6 s. Podobné maximum v prvním měření se vyskytovalo také ve fázi SVD (srovnání graficky znázorněno na obrázku B.2 v příloze) a iterace (graf B.3 zobrazující srovnání naměřených hodnot této fáze modelu je uveden taktéž v příloze). Je to zřejmě dáno inicializací GPU před prvním měřením, které způsobuje pozorovanou časovou ztrátu. Tento jev se opakoval při nezávislých cyklech měření, vždy při prvním spuštění, je tedy možné předpokládat, že při prvém spuštění výpočtu s pomocí gpuArray na neinicializovaném grafickém procesoru probíhá inicializace grafického procesoru v rámci běhu funkcí nástroje gpuArray.

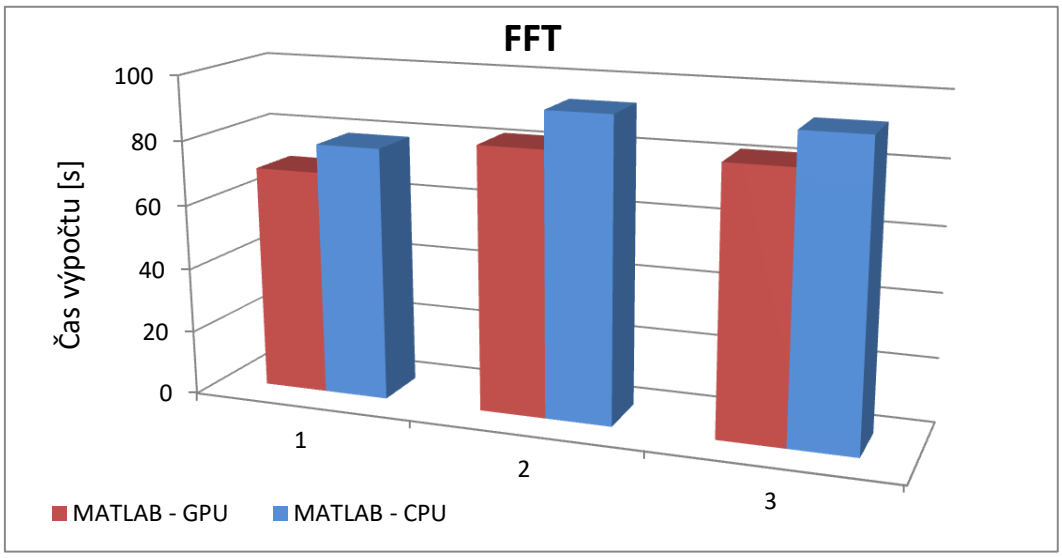


Obr. 5.6: Grafické srovnání délky výpočtu inicializace za použití GPU a CPU.

Fáze inicializace je jediná, jejíž paralelizace s pomocí nástroje gpuArray nepřinesla časovou úsporu. Je to z důvodu prvotní alokace a nahrávání dat do paměti grafického procesoru v rámci fáze inicializace, která se následně v GPU zpracují. Přenosy dat mezi pamětí grafického procesoru a paměti počítače představují největší časovou ztrátu mezi operacemi, prováděnými pomocí grafického akcelerátoru.

Jediná fáze rekonstrukčního modelu, u které se jev inicializace grafického procesoru při prvním měření nevyskytoval, je fáze FFT. V této fázi se využívají výpočty

a funkce z předchozí fáze, není třeba tedy alokování dalších polí pro nové mezivýsledky, paměť je již alokována z předchozí fáze. Srovnání naměřených hodnot ve fázi FFT je graficky zobrazeno na obrázku 5.7. Zpracováním výpočtů na grafickém procesoru pomocí gpuArray jsme docílili zrychlení na přibližně 88 % původní doby trvání výpočtů.



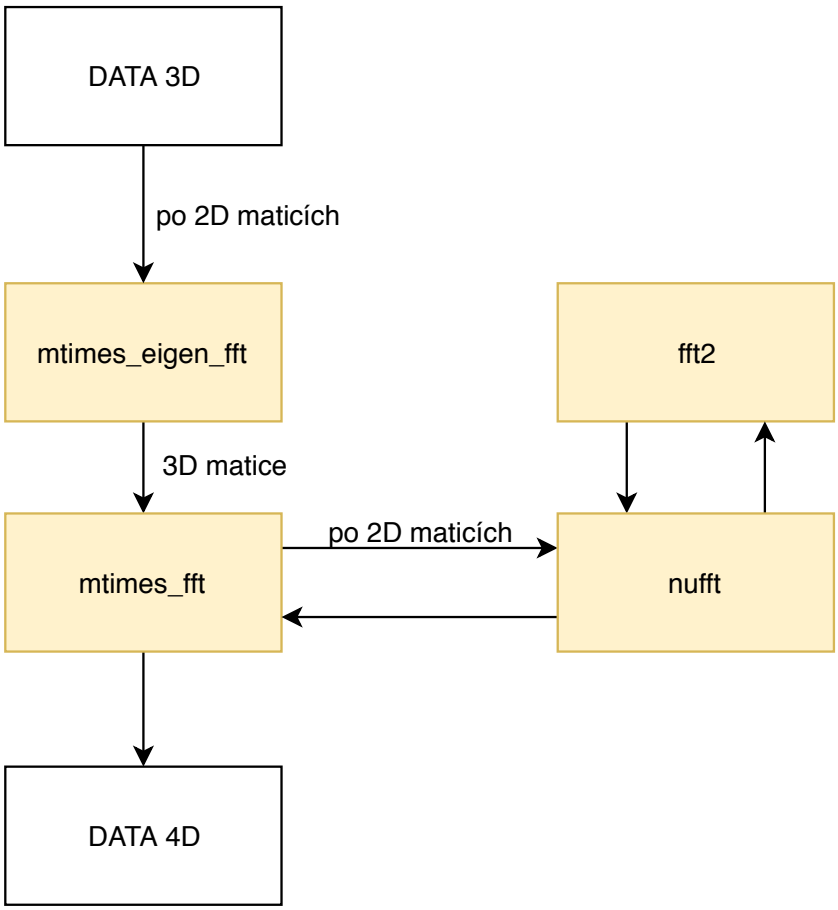
Obr. 5.7: Grafické srovnání délky výpočtu FFT za použití GPU a CPU.

Nástroj gpuArray je navržen pro jednoduché použití v programu MATLAB. Neumožňuje tedy nastavení podrobnějších parametrů, z čehož plyne fakt, že s pomocí gpuArray není možné plně optimalizovat složitější výpočty, jako se ukázalo v případě výpočtů s operátory Fourierovy transformace a citlivosti ve fázi FFT a ve fázi inicializace. Časové úspory je možné dosáhnout i při složitějších výpočtech, ale není tak významná jako v případě použití gpuArray pro výpočet SVD algoritmu. Je tedy spíše vhodnější k použití pro velké objemy dat, na kterých jsou prováděné jednodušší výpočty. Použití gpuArray tímto způsobem se ukázalo jako nejefektivnější.

5.8 Implementace náročných úloh v jazyce Java

Následující oddíl praktické části práce se zabývá optimalizací kódu náročných úloh rekonstrukčního modelu z fáze FFT rekonstrukčního modelu. Funkce, naprogramovaná nízkoúrovňově, tedy za použití primitivních datových typů a pouze jednorozměrných polí, je rychleji zpracována než funkce napsána v programovacím jazyce s vyšší úrovní abstrakce architektury počítače. S touto myšlenkou byly přepracovány funkce náročných úloh rekonstrukčního modelu z fáze FFT rekonstrukčního modelu do podoby metod, jejichž volání je zobrazeno v následujícím schématu 5.8, a spolu

s nimi všechny podřízené funkce pro matematické operace s maticemi komplexních čísel, které jsou v rámci funkcí volány.



Obr. 5.8: Volání metod v implementaci první části gradientního kroku v jazyce Java.

Ve schématu jsou uvedeny názvy klíčových vypracovaných metod v jazyce Java ve třídě `Functions.java` v Java projektu. Volání funkcí kopíruje kaskádu funkcí v rekonstrukčním modelu v MATLABu, vyobrazeném ve schématu 5.3.

Metoda `mtimes_eigen_fft` aplikuje ESPIRiT operátor citlivosti, vypracovaná metoda `mtimes_fft` plní funkci `mtimes.m` z balíčku nástrojů ESPIRiT. Metoda `nufft` odpovídá funkcionalitě stejnojmenné funkce v původním rekonstrukčním modelu, a volá při svém chodu metodu `fft2`, provádějící dvourozměrnou rychlou Fourierovu transformaci, která byla na základě matematického vzorce 2D FFT vypracována v jazyce Java.

Všechny uvedené metody byly otestovány na reálných datech ze skeneru a srovnány s výsledky výpočtů jejich protějšků z rekonstrukčního modelu v MATLABu pro ověření funkcionality.

5.9 Optimalizace funkce NUFFT pomocí knihovny JCuda

Na základě vypracované funkce NUFFs architekturou NVIDIA označovanou kódovým názvemT v jazyce Java, představené v kaskádě metod v předchozí podkapitole, byl aplikován poslední krok z blokového schématu řešení s paralelními výpočty pomocí nástroje JCuda 5.4, a sice převedení výpočtů na procesor grafické karty.

Prvním krokem v paralelizaci za pomocí JCuda bylo přepracování složitějších úloh, vyskytujících se ve funkci NUFFT, do jazyka C.

Druhým krokem bylo vytvořený kód v jazyce C upravit tak, aby mohly funkci zpracovávat paralelní vlákna grafického procesoru. Hlavní překážkou jsou v tomto ohledu cykly `for` a počítadla uvnitř funkcí. Vlákna pracují na sobě nezávisle a nijak se nesynchronizují, není tedy možné udržovat správné pořadí operací pro počítadlo. Bylo třeba tedy cykly `for` eliminovat a nahradit je podmínkou `if` pro umožnění přístupu vláken procesoru. Počítadla musela být odstraněna úplně a algoritmus přepracován tak, aby nebyla potřeba. Výsledný kód odpovídá funkci jádra (anglicky Kernel Function) a je v souborovém formátu CU, neboli ve formátu zdrojového kódu technologie CUDA. Příklad vypracovaného jádra, volaného v paralelizované funkci NUFFT pomocí technologie CUDA a knihovny JCuda, je uveden v příloze C.1. Jedná se o vypracované jádro paralelizované funkce na základě metody v jazyce Java, která je pro srovnání taktéž v příloze C.2.

Třetím krokem pak je vytvoření s funkcemi jádra souvisejících metod v Java třídě `FunctionsParallel.java`, které za pomocí knihovny JCuda volají funkce jádra a provádí potřebné nastavení. To zahrnuje inicializaci funkce jádra, překlad do souboru formátu PTX [2], který je při spuštění grafickým procesorem přeložen na binární kód, dále alokaci vstupních a výstupních dat a nastavení parametrů jádra. Metody v Javě s pomocí knihovny JCuda také zajišťuje přesun dat z grafického procesoru zpět do paměti počítače a je možné s výsledkem v Java programu dále pracovat.

5.9.1 Implementace NUFFT

Metoda `nufft`, vykonávající výpočet NUFFT, byla díky modularizaci volaných funkcionalit rozdělena na důležité výpočetní části již při implementaci v jazyce Java. Tyto funkcionality následně byly přepracovány do jazyku C.

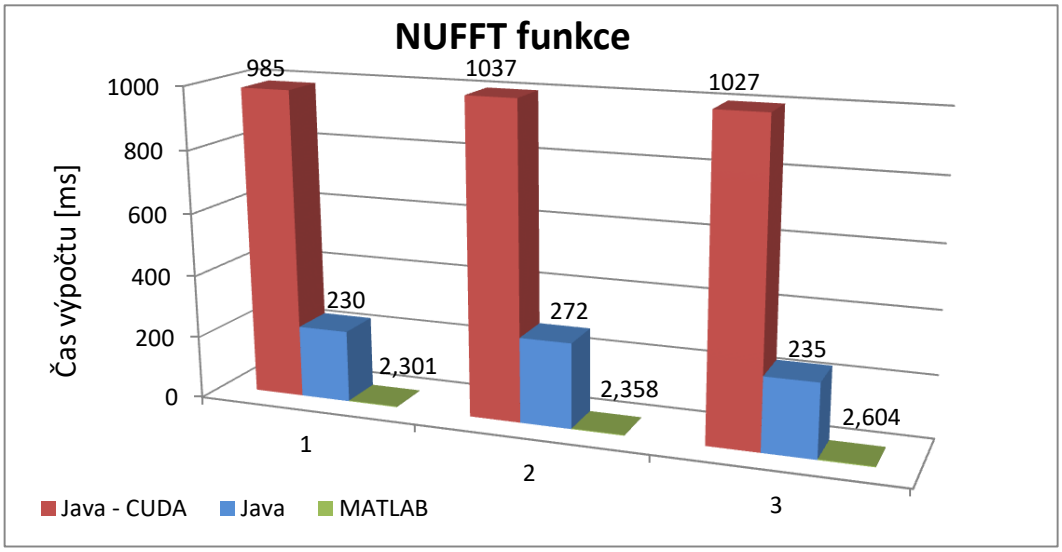
Jedna z vypracovaných metod, které vykonávají operace s maticemi komplexních čísel, `multiplicate2dMatrices`, byla po přepracování do jazyku C shledána nevhodnou pro paralelizaci. Kód této metody v jazyce C je uveden pro ilustraci

[2] zkratka z anglického Parallel Thread Execution - spuštění paralelních vláken

v příloze C.3. Nevhodnost metody pro paralelizaci pomocí technologie CUDA pramení ze závislostí mezivýsledků jeden na druhém mezi probíhajícími cykly výpočtů. Jelikož však vlákna pracují na sobě nezávisle, bez jejich synchronizace není možné předávat výsledky sekvenčně, jak je pro výpočet v metodě potřeba. Kdybychom této synchronizace docílili, ztratila by se výhoda datového paralelismu, který umožňuje výpočty na nezávislých částech dat provádět současně. Vlákna by na sebe musela čekat, a tím by se navyšovala časová ztráta.

5.9.2 Naměřené délky výpočtů NUFFT

Po optimalizaci funkce NUFFT s pomocí technologie CUDA a knihovny JCuda byla provedena sada měření délky výpočtu funkce v hlavním programu testovací třídy `RunDebug.java` v Java projektu. Měření bylo provedeno pro srovnání také v MATLABu, a sice délky výpočtu funkce `nufft.m` součástí rekonstrukčního modelu, a v Java programu, kde byla změřena délka výpočtu vypracované metody `nufft` v jazyce Java. Měření byla opakovaně provedena pro počet 12 projekcí na snímek. Výsledky měření výpočtů, provedených na datech s nastavením 12 projekcí na snímek, pak byly zaneseny do grafického srovnání 5.9 jednotlivých implementací funkce NUFFT.



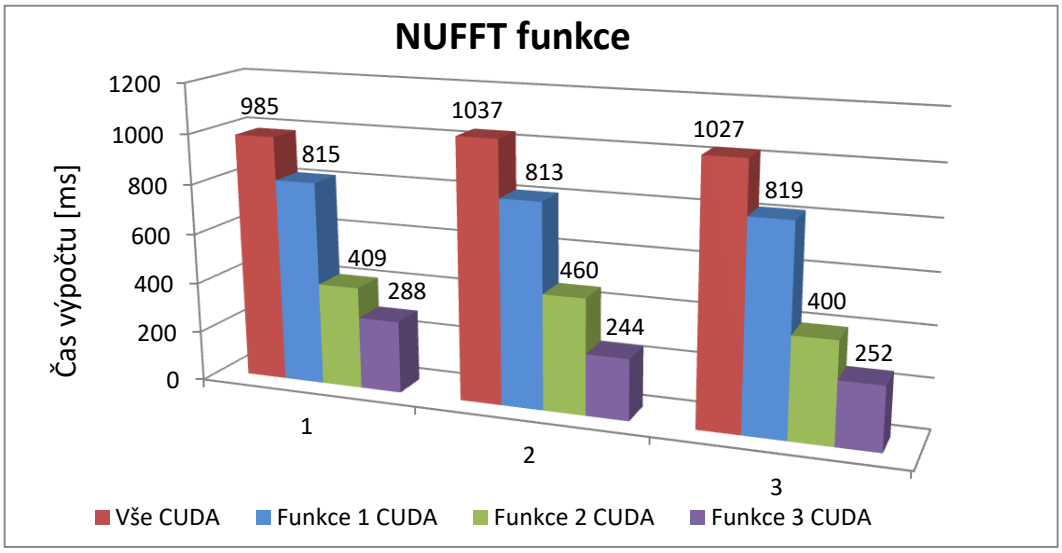
Obr. 5.9: Grafické srovnání délky výpočtu funkce NUFFT v různých implementacích pro data s 12 projekcemi na snímek.

5.9.3 Vyhodnocení výsledků

Z grafu je patrné, že pro takto omezené množství dat režie inicializace a volání funkcí jader výrazně převyšuje časovou úsporu, získanou paralelizací dat. Pro operace s ma-

ticemi komplexních čísel se jako nejvhodnější pro menší objemy dat ukazuje jazyk MATLAB, který byl primárně vyvíjen pro práci s maticemi, jak také napovídá jeho název MATLAB – zkratka z anglického Matrix Laboratory[3][22]. Funkce NUFFT v MATLABu probíhala průměrně 2,4 milisekund. Spuštění implementace funkce NUFFT v jazyce Java bez paralelního zpracování dat trvalo průměrně 245,7 ms. Nejméně časově efektivní byl výpočet NUFFT v podobě optimalizované funkce NUFFT s pomocí technologie CUDA a knihovny JCuda, výpočet trval průměrně 1016,3 ms.

Pro srovnání časové efektivnosti jednotlivých funkcí jádra bylo provedeno měření délky výpočtu NUFFT, kdy postupně ve funkci NUFFT byla použita vždy jen jedna funkce jádra a zbytek byl nahrazen odpovídajícími neparalelizovanými metodami v Javě. Naměřené hodnoty byly srovnány následně mezi sebou a s maximálně paralelizovanou verzí v grafu 5.10.



Obr. 5.10: Grafické srovnání délky výpočtu funkce NUFFT v různých implementacích pro data s 12 projekcemi na snímek.

Funkce 1 odpovídá Java metodě `fft2Parallel`, která při spuštění volá funkci jádra `JCudaFFT2Base.cu`, a je ve srovnání s ostatními funkcemi nejméně časově efektivní. Je to dáno faktem, že se v metodě volá funkce jádra dvakrát, pro jednotlivé rozměry vstupní matice.

Funkce 2 na obrázku odpovídá funkci jádra `JCudaTransposeCplx.cu`, volané metodou `transposeNotConjugate2dMatrixParallel`. Je časově efektivnější než předchozí funkce, ale méně efektivní než **funkce 3**.

[3]neboli maticová laboratoř

Funkce 3 v grafu odpovídá funkci jádra `JCudaElByEl2d.cu`, volané metodou `elByEl2dMultiBnonComplexParallel`. Je to nejjednodušší funkcionalita, pracuje jen s jedním rozměrem matice, a tedy i jedním blokem vláken, je proto nejrychlejší.

6 Závěr

Cílem diplomové práce bylo nastudování principu zobrazování v magnetické rezonanci, seznámení se s dodaným rekonstrukčním modelem a jeho následná optimalizace s pomocí grafického akcelerátoru. Stěžejní částí práce se měla zabývat optimalizací výpočetně náročných úloh rekonstrukčního modelu v MATLABu s pomocí funkce gpuArray a dále na základě zadaného rekonstrukčního modelu měla být vypracována funkce NUFFT s pomocí technologie CUDA a knihovny JCuda. Optimalizovaná funkce měla být spuštěna na reálných datech z MR skeneru a výsledky srovnány a vykresleny do grafů.

V diplomové práci byla popsána technologie magnetické rezonance z hlediska principu, využití a sběru dat pro MRI. Byl následně rozebrán princip rekonstrukce obrazu z MR, a to statické i dynamické, a představeny proximální algoritmy používané pro rekonstrukci obrazu v MRI. Dále byl analyzován rekonstrukční model a rozebrán časově náročný algoritmus SVD. Představeny byly rovněž možnosti paralelizace výpočtů, typy paralelismu, popsán rozdíl mezi zpracováním dat na procesoru počítače a grafickém procesoru a vysvětlena abstrakce paměti v grafických kartách NVIDIA. Dále se práce zabývala platformou CUDA pro vývoj softwaru, architekturou grafického procesoru a základními příkazy v prostředí CUDA.

Výstupem práce je návrh systému pro paralelní zpracování dat včetně blokového schématu systému a následně realizace paralelizovaných výpočtů na grafické kartě. V práci jsou uvedeny výpočetní prostředky z hlediska hardwaru i softwaru, který byl použit k vypracování praktické části práce. Představen je především soubor nástrojů pro vývoj programů pro grafický procesor JCuda a nástroj gpuArray, jež byly v realizaci řešená použity k práci s GPU, a také vstupní data, se kterými rekonstrukční model pracuje. Dále je podrobně popsán rekonstrukční model v MATLABu a jeho fáze, které v rámci rekonstrukci probíhají.

Prvním vypracovaným řešením je paralelizovaný algoritmus SVD, který byl podroben měřením rychlosti výpočtu na reálných datech z MR skeneru a srovnán s měřeními rychlosti SVD funkce rekonstrukčního modelu. Získané výsledky ukázaly, že paralelní zpracování dat v MATLABu pomocí gpuArray více než dvojnásobně zrychlilo původní algoritmus. Vypracovanou implementací algoritmu v jazyce Java, využívající taktéž paralelních výpočtů na GPU pomocí knihovny JCuda, jsme časové úspory nedosáhli, výpočet trval déle než neparalelizovaná funkce na výpočet SVD z důvodu použití neoptimalizované metody knihovny cuSOLVER, která vykazuje časové ztráty kvůli častým přenosům dat mezi počítačem a grafickým procesorem.

Stěžejní částí práce byla optimalizace rekonstrukčního modelu v MATLABu pomocí funkce gpuArray. Časové úspory jsme díky paralelním výpočtům na grafickém procesoru dosáhli, zhruba 1,15krát. Zvýšení časové efektivity výpočtů však není

natolik výrazné jako při použití gpuArray pro výpočet algoritmu SVD. Nástroj gpuArray je navržen pro jednoduché použití v programu MATLAB, neumožňuje tak nastavení podrobnějších parametrů. Nástrojem gpuArray není proto možné plně optimalizovat složitější výpočty a je nejvhodnější k použití pro velké objemy dat, na kterých jsou prováděné jednodušší výpočty.

Poslední částí práce bylo vypracování optimalizované funkce NUFFT pomocí knihovny JCuda a technologie CUDA. Algoritmus byl nejprve přepsán do jazyku Java, a na jeho základě byly výpočetně náročné části přepracovány do zdrojového kódu technologie CUDA. V programu Java je pak s pomocí knihovny JCuda v rámci funkce NUFFT volán tento zdrojový kód, který umožňuje předání instrukcí a parametrů pro výpočet na grafické kartě. Optimalizovaná funkce NUFFT poté byla spuštěna na reálných datech z MR skeneru a výsledky srovnány a vykresleny do grafů.

Výsledky měření délky výpočtů optimalizované NUFFT funkce s pomocí technologie CUDA na testovaných datech nedosáhly časové úspory ve srovnání s rekonstrukčním modelem. Je to dáno omezeným množstvím dat, které do funkce vstupují, a tudíž není možné dosáhnout takové míry datového paralelismu, aby časová úspora paralelního zpracování dat vyvážila nároky pro spuštění funkcí technologie CUDA. Dalším důvodem jsou omezené výpočetní možnosti grafického procesoru sekundární grafické karty, která byla pro testování použita, ve srovnání s větší pamětí a větší propustností použitého procesoru počítače. A v poslední řadě je nedostatečná časová úspora dána faktem, že náročné výpočty ve funkci NUFFT na sebe navazují. Není proto možné využít úlohového paralelismu, ale jen datového. Proto paralelizace jednoduššího algoritmu SVD byla výrazně efektivnější než paralelizace funkce NUFFT, jejíž náročné paralelizované úlohy musely být spuštěny sekvenčně.

Literatura

[1] BUSHONG, Stewart C. a Geoffrey D. CLARKE. *Magnetic resonance imaging: physical and biological principles.* 5. vydání. St. Louis: Elsevier Mosby, 2014, 513 stran. ISBN 9780323073547.

[2] BROWN, Robert W., Yu-Chung N. CHENG, E. Mark HAACKE, Michael R THOMPSON a Ramesh VENKATESAN. *Magnetic resonance imaging: physical principles and sequence design.* 2. vydání. Hoboken, New Jersey: John Wiley & Sons, Inc., 2014. ISBN 978-047-1720-850.

[3] RAJMIC, Pavel. *Brief introduction into the MR acquisition and reconstruction.* Brno: Vysoké učení technické v Brně, 2018.

[4] HALL, Matt G. Continuity, the Bloch-Torrey equation, and Diffusion MRI. *arXiv.org* [online]. Srpen 2016, 41 stran [cit. 2018-11-04]. Bibliografický kód 2016arXiv160802859H. Dostupné z URL: <https://arxiv.org/abs/1608.02859>.

[5] PASCHAL, Cynthia B. a H. Douglas MORRIS. K-space in the clinic. *Journal of Magnetic Resonance Imaging* [online]. 2004, roč. 19, č. 2, s. 145–159 [cit. 2018-11-04]. DOI: 10.1002/jmri.10451. ISSN 1053-1807. Dostupné z URL: <http://doi.wiley.com/10.1002/jmri.10451>.

[6] DESHMANE, Anagha; GULANI, Vikas; GRISWOLD, Mark A.; aj. Parallel MR Imaging. *Journal of Magnetic Resonance Imaging* [online]. International Society for Magnetic Resonance in Medicine, 2012, roč. 36, č. 1, s. 55–72 [cit. 2018-11-02]. ISSN 1522-2586. Dostupné z URL: <https://onlinelibrary.wiley.com/doi/epdf/10.1002/jmri.23639>.

[7] JAN, Jiří. *Medical image processing, reconstruction, and restoration: concepts and methods*. Boca Raton, FL: Taylor & Francis, 2006. ISBN 978-0-8247-5849-3.

[8] LUSTIG, M., D.L. DONOHO, J.M. SANTOS a J.M. PAULY. Compressed Sensing MRI. *IEEE Signal Processing Magazine* [online]. 2008, roč. 25, č. 2, s. 72–82 [cit. 2018-11-12]. DOI: 10.1109/MSP.2007.914728. ISSN 1053-5888. Dostupné z URL: <http://ieeexplore.ieee.org/document/4472246/>.

[9] HU, Yanchun, Song GAO, Liuquan CHENG a Shanglian BAO. Review: K-space trajectory development. *2013 IEEE International Conference on Medical Imaging Physics and Engineering* [online]. IEEE, 2013, s. 356–360 [cit. 2018-11-13]. DOI: 10.1109/ICMIPE.2013.6864568. ISBN 978-1-4673-5887-3. Dostupné z URL: <http://ieeexplore.ieee.org/document/6864568/>.

[10] LANDINI, Luigi, Vicenzo POSITANO a Maria Filomena SANTARELLI. *Advanced image processing in magnetic resonance imaging*. Boca Raton, FL: Taylor & Francis, 2005. ISBN 9780824725426.

[11] OTAZO, Ricardo, Daniel K. SODICKSON a Emmanuel CANDÈS. Low-rank plus sparse matrix decomposition for accelerated dynamic MRI with separation of background and dynamic components. *Magnetic Resonance in Medicine* [online]. 2014, roč. 73, č. 3, s. 1125–1136 [cit. 2019-04-29]. DOI: 10.1002/mrm.25240. ISSN 07403194. Dostupné z URL: <https://onlinelibrary.wiley.com/doi/epdf/10.1002/mrm.25240>.

[12] SADEK, Rowayda A. SVD Based Image Processing Applications: State of The Art, Contributions and Research Challenges. *International Journal of Advanced Computer Science and Applications* [online]. 2012, roč. 3, č. 7, s. 26–34 [cit. 2018-12-02]. DOI: 10.14569/IJACSA.2012.030703. ISSN 2158107X. Dostupné z URL: <https://arxiv.org/abs/1211.7102>.

[13] HAGER, Georg a Gerhard WELLEIN. *Introduction to high performance computing for scientists and engineers*. Boca Raton, FL: CRC Press, 2011. ISBN 978-1-4398-1192-4.

[14] KREWELL, Kevin. What's the Difference Between a CPU and a GPU? In: NVIDIA. *The NVIDIA Blog* [online]. 16. prosinec 2009 [cit. 2018-11-25]. Dostupné z URL: <https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/>.

[15] REGE, Ashu. *An Introduction to Modern GPU Architecture* [online]. Santa Clara (California): NVIDIA Corporation, 2008 [cit. 2018-11-25]. Dostupné z URL: <http://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf>.

[16] CUDA C Programming Guide. *CUDA Toolkit Documentation* [online]. Santa Clara (California): NVIDIA Corporation, 30. říjen 2018 [cit. 2018-11-26]. Dostupné z URL: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>.

[17] SHEN, Jie, Diego VELA, Ankita SINGH, Kexing SONG, Guoshang ZHANG, Bradon LAFRENIERE a Hao CHEN. GPU/CPU parallel computation of material damage. *Engineering with Computers* [online]. 2015, roč. 31, č. 3, s. 647–660 [cit. 2018-11-23]. DOI: 10.1007/s00366-014-0367-9. ISSN 0177-0667. Dostupné z URL: <http://link.springer.com/10.1007/s00366-014-0367-9>.

[18] jcuda.org. *jcuda.org. – Java bindings for CUDA* [online]. 2009–2018 [cit. 2018-11-27]. Dostupné z URL: <http://www.jcuda.org/>.

[19] MathWorks. *Array stored on GPU – MATLAB gpuArray* [online]. 1994–2019 [cit. 2018-03-10]. Dostupné z URL: <https://uk.mathworks.com/help/distcomp/gpuarray_object.html>.

[20] Orange Owl Solutions, GitHub. *Linear-Algebra/SVD_Matlab.m. – OrangeOwlSolutions – GitHub* [online]. 2017 [cit. 2018-12-08]. Dostupné z URL: <https://github.com/OrangeOwlSolutions/Linear-Algebra/blob/master/SVD/SVD_Matlab.m>.

[21] cusolverDnCgesvd performance vs MKL. In: *NVIDIA Developer Forums* [online]. Santa Clara (California): NVIDIA Corporation, 5. červenec 2015 [cit. 2019-05-03]. Dostupné z URL: <https://devtalk.nvidia.com/default/topic/830894/cusolverdncgesvd-performance-vs-mkl/>.

[22] MathWorks. *Matrices and Arrays – MATLAB & Simulink* [online]. 1994–2019 [cit. 2019-05-12]. Dostupné z URL: <https://www.mathworks.com/help/matlab/learn_matlab/matrices-and-arrays.html>.

Seznam symbolů, veličin a zkratek

γ	gyromagnetický poměr [Hz/T]
ALU	aritmeticko-logická jednotka – Arithmetic Logic Unit
API	programovací rozhraní aplikací – Application Programming Interface
B	magnetická indukce [T]
CPU	centrální procesorová jednotka – Central Processing Unit
CT	počítačová tomografie – Computer Tomography
CUDA	Compute Unified Device Architecture
DRAM	dynamická paměť s přímým přístupem – Dynamic Random Access Memory
f	kmitočet [Hz]
FFT	rychlá Fourierova transformace – Fast Fourier transform
FID	Free Induction Decay
FT	Fourierova transformace – Fourier transform
GPU	grafický procesor – Graphics Processing Unit
$^1\mathbf{H}$	atom vodíku-1 – Protium
ID	identifikátor nebo taky identifikační číslo
L+S model	model matice s nízkou hodností a řídké matice – low-rank plus sparse model
MATLAB	Matrix Laboratory
MR	magnetická rezonance – Magnetic Resonance
MRI	zobrazování magnetickou rezonancí – Magnetic Resonance Imaging
NUFFT	neuniformní rychlá Fourierova transformace – Non-Uniform Fast Fourier transform
NVCC	NVIDIA CUDA kompilátor – NVIDIA CUDA Compiler
PC	osobní počítač – Personal Computer
PET	pozitronová emisní tomografie – Positron Emission Tomography
RAM	paměť s přímým přístupem – Random Access Memory
RF	vysokofrekvenční – Radio requency
SNR	odstup signálu od šumu – Signal-to-Noise Ratio
SVD	rozklad na singulární hodnoty – Singular Value Decomposition
$\mathbf{T}_1$	relaxační čas T_1 (relaxace spin–mřížka)
$\mathbf{T}_2$	relaxační čas T_2 (relaxace spin–spin)
TV	totální variace – Total Variation

Seznam příloh

A Proximálně gradientní algoritmus **56**

B Výsledky měření rychlosti výpočtů implementace rekonstrukčního modelu s pomocí gpuArray **57**

B.1 Naměřené hodnoty délky výpočtů s pomocí gpuArray 57

B.2 Grafická zobrazení srovnání rychlosti výpočtů s pomocí gpuArray a zadaného rekonstrukčního modelu 58

C Ukázkové kódy **60**

D Obsah přiloženého CD **63**

A Proximálně gradientní algoritmus

Zápis proximálně gradientního algoritmu pro řešení L+S modelu dynamické rekonstrukce magnetické rezonance s urychlením konvergence (převzato z [3]).

Vstupní informace: {Data d, operátory $\mathcal{F}, \mathcal{F}^*, \mathcal{C}, \mathcal{C}^*$; konstanty λ_L, λ_S; proximální operátory odvozené od r_L a r_S}
Výstupní data: $\{M^{(n)}; L^{(n)}; S^{(n)}\}$
Počáteční nastavení hodnot: {
$t > 0$;
$M^{(0)} = L^{(0)} + S^{(0)}$;
$n = 0$; }
Iterace, dokud nedosáhneme konvergence výsledků: {
$M^{(n+1)} = M^{(n)} - t \cdot \mathcal{C}^*\mathcal{F}^*(\mathcal{F}\mathcal{C}(M^{(n)}) - d)$
$L^{(n+1)} = \text{prox}_{t\lambda_L r_L}(M^{(n+1)} - S^{(n)})$
$S^{(n+1)} = \text{prox}_{t\lambda_S r_S}(M^{(n+1)} - L^{(n+1)})$
$M^{(n+1)} = L^{(n+1)} + S^{(n+1)}$
$n = n + 1$ }

B Výsledky měření rychlosti výpočtů implementace rekonstrukčního modelu s pomocí gpuArray

B.1 Naměřené hodnoty délky výpočtů s pomocí gpuArray

Výpis B.1: Příklad výpisu do konzole v programu MATLAB z měření délky výpočtů fází rekonstrukčního modelu s pomocí nástroje gpuArray.

```
Elapsed time is 96.112336 seconds.
inicializace
Elapsed time is 307.840085 seconds.
fft
Elapsed time is 69.875796 seconds.
svd
Elapsed time is 84.013390 seconds.
1. iterace
Elapsed time is 171.125462 seconds.
```

Tab. B.1: Srovnání výpočtů fáze inicializace rekonstrukčního modelu v MATLABu mezi původním programem na CPU a optimalizovanou implementací na GPU.

Použitá implementace	Čas 1 [s]	Čas 2 [s]	Čas 3 [s]
Výpočet na CPU (původní model)	67,648	69,797	71,316
Výpočet na GPU (gpuArray)	307,84	108,809	106,643

Tab. B.2: Srovnání výpočtů fáze FFT rekonstrukčního modelu v MATLABu mezi původním programem na CPU a optimalizovanou implementací na GPU.

Použitá implementace	Čas 1 [s]	Čas 2 [s]	Čas 3 [s]
Výpočet na CPU (původní model)	78,659	93,477	92,283
Výpočet na GPU (gpuArray)	69,876	81,919	81,903

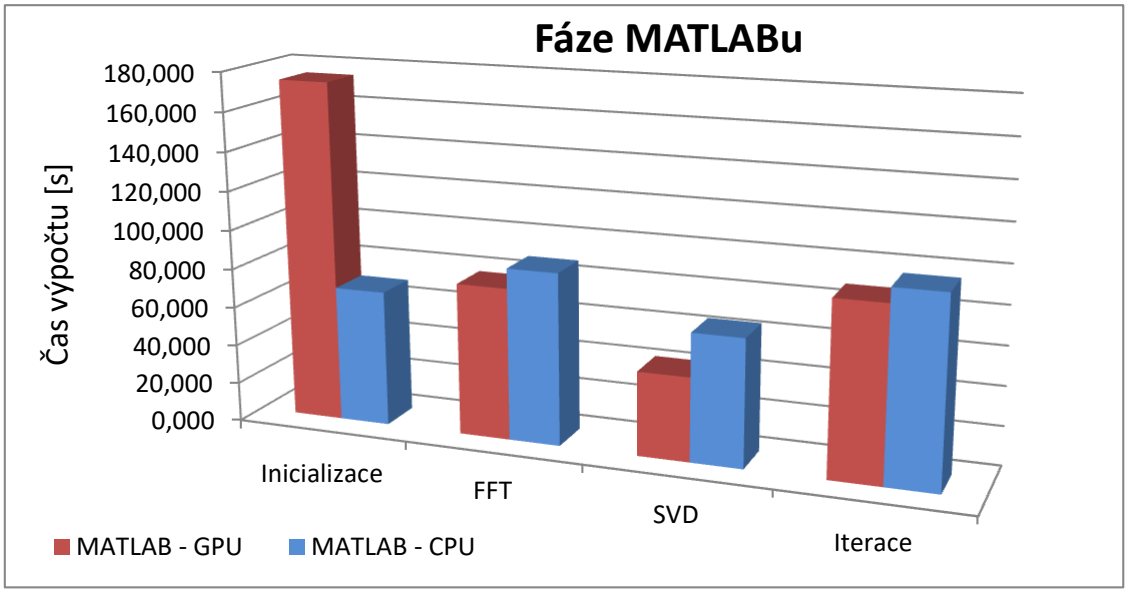
Tab. B.3: Srovnání výpočtů fáze SVD rekonstrukčního modelu v MATLABu mezi původním programem na CPU a optimalizovanou implementací na GPU.

Použitá implementace	Čas 1 [s]	Čas 2 [s]	Čas 3 [s]
Výpočet na CPU (původní model)	65,304	67,66	62,478
Výpočet na GPU (gpuArray)	84,013	24,301	20,281

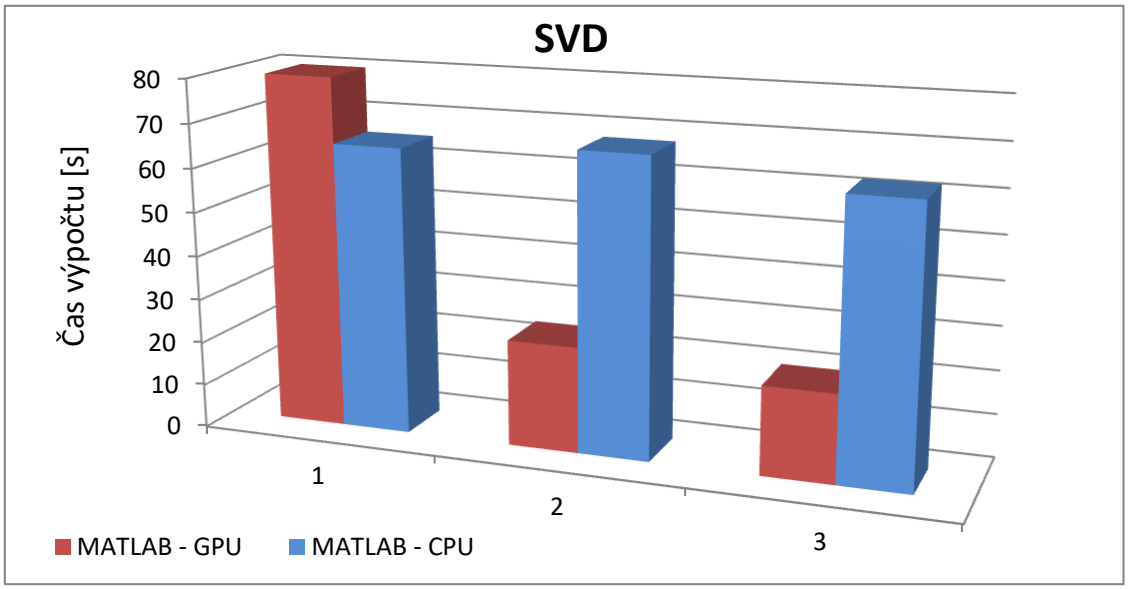
Tab. B.4: Srovnání výpočtů fáze iterace rekonstrukčního modelu v MATLABu mezi původním programem na CPU a optimalizovanou implementací na GPU.

Použitá implementace	Čas 1 [s]	Čas 2 [s]	Čas 3 [s]
Výpočet na CPU (původní model)	97,651	99,291	91,895
Výpočet na GPU (gpuArray)	171,125	50,488	44,32

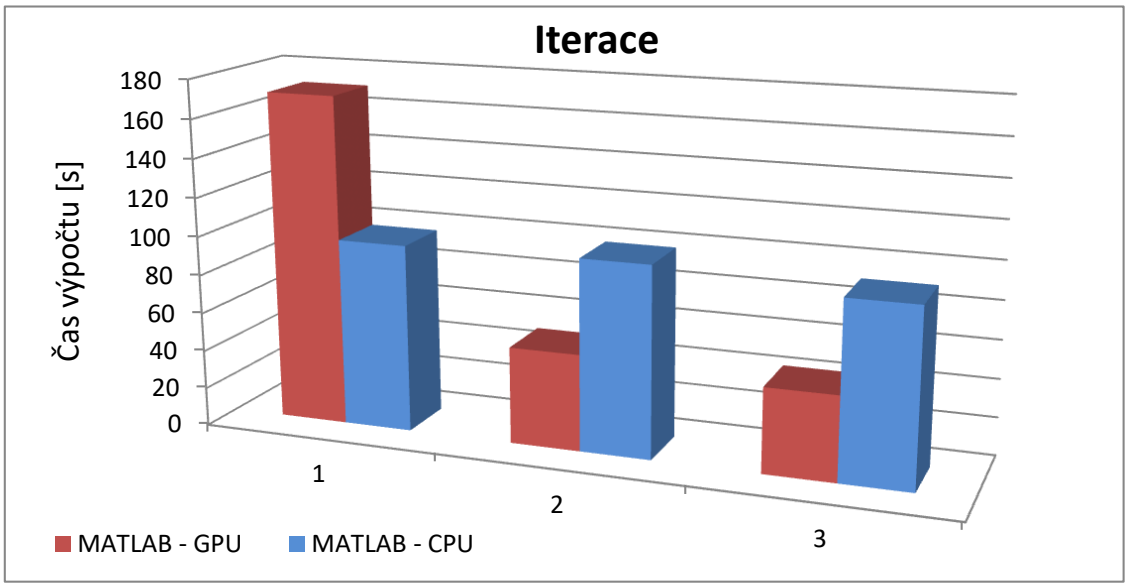
B.2 Grafická zobrazení srovnání rychlosti výpočtů s pomocí gpuArray a zadaného rekonstrukčního modelu



Obr. B.1: Grafické srovnání rychlosti výpočtu inicializace za použití GPU a CPU.



Obr. B.2: Grafické srovnání rychlosti výpočtu inicializace za použití GPU a CPU.



Obr. B.3: Grafické srovnání rychlosti výpočtu iterace za použití GPU a CPU.

C Ukázkové kódy

Výpis C.1: Funkce jádra pro výpočet násobení matice komplexních čísel a matice čísel reálných ve formě 1D vektoru prvek po prvku v jazyce C.

```
extern "C"
__global__ void elByEl2dMultiBnonComplex(float *A, float *B,
                      float *res, int res_size) {

  // Alokace indexu pro paralelni vypocet
    int i = threadIdx.x + blockIdx.x * blockDim.x;

  // Vypocet nasobeni matic prvek po prvku
    if (i < res_size) {
        if (i == 0) {
            res[i] = A[i] * B[i];
            res[i+1] = A[i + 1] * B[i];
        } else if (i%2 == 0) {
            res[i] = A[i] * B[i/2];
            res[i+1] = A[i + 1] * B[i/2];
        }
    }
}
```

Výpis C.2: Metoda pro výpočet násobení matice komplexních čísel a matice čísel reálných ve formě 1D vektoru prvek po prvku v jazyce Java.

```
public static float[] elByEl2dMultiBnonComplex(float[] A,
                          int[] Indices, float[] B) {

    // Indexy 2d matice A
    int m = Indices[0];
    int n = Indices[1];

    //
    float[] res = new float[A.length];
    int resCounter = 0;
    int counter = 0;

    // Nasobeni prvek po prvku:
    for (int j = 0; j < m; j++) {
      for (int i = 0; i < n; i++) {
        res[resCounter++] = A[2 * j * n + 2 * i] * B[counter];
        res[resCounter++] = A[2 * j * n + 2 * i + 1] *
                                                B[counter++];
      }
    }

    return res;
  }
```

Výpis C.3: Přepracovaná metoda pro výpočet násobení dvou matic komplexních čísel ve formě 1D vektoru v jazyce C.

```
void multiplicate2dMatrices(double *A, int *aIndices,
          double *Btransposed, int *bIndices, double *res) {
    // Definice dimenzi matic
    int aDim1 = aIndices[0];
    int aDim2 = aIndices[1];
    int bDim1 = bIndices[0];

    // Vypocet nasobeni matic komplexnich cisel
    for (int k = 0; k < aDim1; k++) {
        for (int j = 0; j < bDim1; j++) {
            for (int i = 0; i < aDim2; i++) {
                // Realna cast komplexniho cisla
                res[2 * k * bDim1 + 2 * j] +=
                  A[2 * i + k * aDim2 * 2] *
                  Btransposed[2 * i + j * aDim2 * 2] -
                  A[2 * i + 1 + k * aDim2 * 2] *
                  Btransposed[2 * i + 1 + j * aDim2 * 2];

                // Imaginarni cast komplexniho cisla
                res[2 * k * bDim1 + 2 * j + 1] +=
                  A[2 * i + 1 + k * aDim2 * 2] *
                  Btransposed[2 * i + j * aDim2 * 2] +
                  A[2 * i + k * aDim2 * 2] *
                  Btransposed[2 * i + 1 + j * aDim2 * 2];
            }
        }
    }
}
```

D Obsah přiloženého CD

- Adresář *Java* obsahuje soubory naprogramovaného řešení v jazyce Java s pomocí systémové knihovny JDK verze 10.0.2.
  - podaresář *.settings* obsahuje nastavení Java projektu
  - podaresář *src* obsahuje zdrojové soubory projektu s příponou .java
  - podaresář *lib* obsahuje použité Java knihovny
  - podaresář *target* obsahuje zkompilované soubory Java tříd projektu s příponou .class
  - soubory .classpath a .project jsou pomocné soubory Java projektu
  - soubory s příponou .cu jsou vypracované funkce jádra technologie CUDA
  - soubor pom.xml je seznam závislostí nástroje Maven, použitého v projektu
- Adresář *Matlab* obsahuje soubory paralelizovaného rekonstrukčního modelu v jazyce MATLAB s pomocí nástroje gpuArray na základě zadaného rekonstrukčního modelu. Řešení bylo vypracováno v MATLABu verze R2017b 9.3.0.
  - podaresář *CS* obsahuje hlavní výpočetní funkci rekonstrukce MRI
  - podadresář *toolboxy* obsahuje funkce balíčku nástrojů ESPIRiT
  - soubor *runCS_rekonstrukce_dynamika.m* je určený pro spuštění implementace rekonstrukčního modelu
- Soubor *DP_bijotova.pdf* obsahuje diplomovou práci